ELSEVIER

Theoretical Computer Science 269 (2001) 363–417

Theoretical Computer Science

www.elsevier.com/locate/tcs

A declarative framework for object-oriented programming with genetic inheritance

Joaquín Mateos Lago *, Mario Rodríguez Artalejo [1]

Departamento de Sistemas Informáticos y Programación. Universidad Complutense de Madrid, Avda. Complutense s/n. E-28040 Madrid, Spain

Received September 1999; revised July 2000; accepted November 2000
Communicated by G. Levi

Abstract

Seeking the integration of the object-oriented and declarative programming paradigms offers advantages for the software life-cycle activities. Specification is benefited from using declarative expressions as functional descriptions of components, enjoying formal semantic models. But the integration of both paradigms, object-oriented and declarative, following a translation scheme sets an unavoidable *representation distance*. Classes, inheritance, attributes and methods are codified with abstract elements, thus not being primitive. This work aims to offer a declarative formal model where the main features of object-oriented programming are nuclear, focusing in an algebraic formalization of purely functional objects. Substantially extending (Mateos-Lago and Rodríguez-Artalejo, PLILP'96, Lecture Notes in Computer Science, Vol. 1140, Springer, Berlin, 1996, pp. 62–76), we include operations to homogeneously model methods and class-external functions. Multiple inheritance is supported and extended with *genetic inheritance* and expressions are flexibly typed using *genome typing*. Following (González-Moreno et al., J. Logic Programming 40(1) (1999) 47), we use a rewriting logic as a technical tool that helps to formalize the semantics based on continuous algebras (Goguen et al., J. ACM 24(1) (1977) 68), and we show initiality with the existence of a distinguished model for program semantics.

Keywords: Object-oriented declarative programming; Genetic inheritance; Algebraic semantics; Paradigm integration

1. Introduction

Regardless other virtues related with software reuse, the object-oriented programming paradigm justifies itself in terms of problem solving advantages. As [4] put it:

* Corresponding author.
E-mail addresses: jmlago@sip.ucm.es (J. Mateos Lago), mario@sip.ucm.es (M. Rodríguez Artalejo).
[1] Partially supported by Spanish CICYT Project TREND (TIC98-0445-C03-02).

0304-3975/01/$ - see front matter © 2001 Published by Elsevier Science B.V.
PII: S0304-3975(01)00013-5

"One powerful design strategy, which is particularly appropriate to the construction of programs for modeling physical systems, is to base the structure of our programs on the structure of the system being modeled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model". Not only physical systems reduce to the object paradigm, but many transactional, and even symbolic information systems, benefit from being cast through the object-oriented perspective.

On a similar basis, declarative programming has shown to be a natural paradigm to reason about symbolic concepts, offering a higher abstraction level that eases the task of programming. Software specification is also benefited from using declarative expressions as functional descriptions for system components. Besides, declarative programming enjoys formal semantic models based on different theoretical frameworks allowing clean, clear semantics for programs.

It seems an interesting step to seek the integration of both paradigms, object-oriented and declarative, looking to combine their respective advantages. A research in integrative computation models has come along in an effort to represent object-oriented features inside various logical frameworks like logic programming [14, 24, 36], dynamic logic [42], linear logic [8, 15], functional-logic programming [6], concurrent constraint programming [23], algebraic settings [16, 17, 26], rewriting logic [33], type theory [10, 25, 37], and the typed λ-calculus [12]. Independently of their degree of success, working in a translation scheme sets an unavoidable *representation distance* [35] between the nuclear object-oriented concepts and the semantic elements of the declarative framework which codify the former.

The inspiring thought of this work is to reduce the representation distance for concepts like classes, attributes, and inheritance which must play a primitive role. Our aim is to give a new formal model for the main features of object-oriented programming, using techniques taken from the fields of algebraic specification, functional programming, and denotational semantics. In this sense we view objects as primitives like in the ς-calculus of [1, 2]. Distinctly, we focus in an algebraic formalization to present purely functional objects with a general inheritance scheme, providing a declarative semantics based on the existence of a distinguished model for programs.

This work makes use of concepts like order-sorted algebra [18], feature terms [6], feature trees [7], and records for object state representation. It also adapts the declarative semantics of [20], based on a rewriting logic. As proper characteristics we can anticipate that concepts like classes, attributes, methods and inheritance are considered primitive, multiple inheritance is supported and extended with *genetic inheritance* which subsumes other traditional approaches, expressions are flexibly typed using *genome typing*, recursive objects are allowed, there is non-restricted attribute redefinition and method overriding, method inheritance is formalized by rewriting techniques, and methods and class-external functions are homogeneously represented under the same semantic entity. This flexibility is somewhat diminished by the need of static typing for expressions, entailing a semantics of operations with static dispatch.

Our results can be seen as a substantial extension of the GOTA formalism presented in [30] by incorporating operations to model class methods, as well as the behavior of whatever class-external procedures may occur in a program. Operations in this e*X*tended *GOTA* framework are purely declarative, being defined by rewrite rules. Destructive state update, as found in imperative object-oriented languages, does not take place in XGOTA. We provide a semantics based on continuous algebras [19], which serve to control some aspects of functional computations such as infinite data and lazy evaluation. Following the approach in [20], we use a rewriting logic as a technical tool that helps to formalize the semantics. Our rewriting logic is presented as a calculus that controls, among other things, the proper inheritance of operations. This process is guided by a static notion of well typing.

To follow, Section 2 offers a review of research lines of declarative and object-oriented integration using formal frameworks. To illustrate the expressiveness of operations and genetic inheritance in XGOTA, Section 3 provides some motivating examples. Sections 4 and 5 develop the definitions of XGOTA signatures and algebras and the meaning of expressions. Section 6 introduces the rewrite rules used to define the behavior of operations, and a rewrite calculus to capture the notion of expression reduction. Section 7 deals with models and the model of canonical trees we have devised to show the initial semantics of this formalism, which is formally proved in Section 8. Finally, in Section 9 some conclusions and advances of future work are commented.

2. Review of declarative and object-oriented integration

The objective of this section is to offer a glimpse of other research lines where object-oriented concepts are integrated or represented within a formal declarative semantic framework. We have selected some representative approaches based on different formalisms, with no aim of giving an exhaustive survey. At this moment, our main goal is to show that there is a representation distance between object-oriented concepts like classes, attributes, methods and inheritance and the primitive elements in those settings. Since reducing that distance is our primal concern, this section will provide a motivation.

- *Type-theoretic functional encodings*: Many proposals have been made using the typed lambda-calculus $F^{\omega}_{<:}$, [3, 9, 11, 25, 37] (a comparison can be found in [10]). They focus on encoding objects with λ-expressions using records for attribute distinction, and (type or procedure) abstractions to encapsulate object state and methods (and protect access) via recursive, existential or bounded existential types. The underlying formalism ensures the type discipline and provides for interesting features as polymorphism and higher order. In these approaches, classes are obtained by repeating patterns for object encoding, and inheritance is related to subtyping, since the type of objects decreases in a subtype relation when moving from a class to its subclasses. However, the coexistence of inheritance and subtyping poses some

problems, which lead to impose restrictive conditions on method overriding. Moreover, the contravariant nature of arrow types with respect to the first argument leads to type incompatibilities in the case of binary methods.

- *Overloaded typed λ-calculus*: In contrast to the type-theoretic encodings described above, Ref. [12] is based on a simpler formalism where the state of objects and the methods are separated. Methods are modeled as overloaded functions with several "branches of code", formalized as terms in a so-called $\lambda\&$-calculus. The "branches" correspond to different definitions of the method in various classes. When applying a method to an object, overloading is resolved by choosing the branch whose argument type yields the closest supertype of the run-time type of the object. Inheritance is defined as a relation between classes which corresponds to the subtyping relation for the types of their objects, viewed as records which include attributes but no methods. This approach models multiple inheritance with dynamic dispatch, while avoiding the need to deal with recursive record types or existential types. It works because of strong conditions imposed to overloaded functions, ensuring covariant behavior of the different branches and explicit redefinition in the case of multiple inheritance collisions. These requirements guarantee consistency between method overriding and the subtyping relation. As a consequence, binary methods and multiple dispatch can be easily expressed. Our approach will provide more flexible inheritance and overriding mechanisms, although dynamic dispatch will be lost.
- *Algebraic approaches*: Order-sorted algebra provides for a hierarchy of sorts that can be assimilated to classes [40]. Nevertheless, when initial semantics is pursued the need for different constructor operations in each sort obstructs method inheritance (patterns do not match!). In hidden sorted algebra [16, 17] algebraic semantics is integrated with some concepts from process algebras. Object states are represented by hidden sorts whose details are unknown, while using standard sorts for constructed data. Objects are distinguished by observation of their attributes having values in visible (data) sorts. This leads to a notion of behavioral satisfaction of equations in specifications which provides for a behavioral semantics, intended for gathering possible implementations. Multiple inheritance is limited by the underlying order-sorted setting, and encapsulation becomes hard when having attributes from other hidden sorts (object classes).
- *Coalgebraic approaches*: Coalgebra settings [28] stick to the concept of observation: objects are characterized by their behavior under sequences of operations. Class definitions are represented by sorts with unknown state, attributes are operations selecting (known sort) values, and mutator methods obtain result in the unknown state sort [26]. The specification allows, besides equations, a bisimilarity relation between terms which accounts for observable behavior of the unknown state values. Class implementations are interpreted as coalgebras with a carrier for the state space and a collection of functions for the methods. Cofree (final) semantics is intended for coalgebras where no bisimilar elements exist, representing most economic implementations. Inheritance is forced to preserve types, imposing monotonic attribute and method overriding [27]. The inclusion of class-external functions is

non-homogeneous to the treatment of methods, and methodologically poses some questions about the election of class methods.

- *Constraint + functional schemes*: LIFE [6] is an attempt to integrate functions, relations and types into a single framework. Since basic elements are ψ-terms which are sorted elements described by features, object representation is easily achieved with sorts playing classes and features playing attributes. ψ-terms can be converted into a formula specifying a constraint (conjunction of constraints) about the sort and the features of the element, which can be arbitrarily nested. Its semantics is based on OSF-logic [7] which is a constraint language inside the CLP scheme. Types are defined as the set of values that fulfill a given constraint formula, equivalently the denotation of a ψ-term, and since features are not associated to sorts and ψ-terms are freely constructed, this work reminds of a prototype-based approach [29] rather than a class-based one for object-orientation. Inheritance comes as a relation of subsumption between terms which provides subtyping. Functions are given a weak semantics based on residuation that is semantically incomplete, not having a defined denotation for function symbols [5].
- *Concurrent + constraint + functional schemes*: The Oz computation model [39] is based on constraints, logic satisfiability and concurrent expression reduction. State is represented by so-called cells which are constraints in the computation space, records allow for structuring, while hiding is obtained by the use of fresh variables bound in a constraint to an external name. Procedures are constraints which bind the name to the defining abstraction expression. To encode objects [23], procedures encapsulate state in a local cell, and methods in a record with the names of the corresponding procedure constraints. Classes come as generic objects with a creation method, and multiple inheritance is defined by extending the state cell and method record structures, where a precedence is defined for name clashes. Object protection and concurrent object operation is obtained by the underlying setting, which also provides for higher-order programming and logic variables.

  Maude [33] is based on rewriting logic which formalizes concurrent computation as logical deduction of sequents between terms. Having that terms represent state, sequents provide local changes and deduction shows transition. In this setting, object structures can be defined as terms with external name, class, and attribute values. A pool of objects is built as another structure term including objects. Operations for message passing and method invocation are defined over this pool which apply the existing methods in the corresponding class. Communication events are provided by the rewriting logic which is sound and complete and has initial models.

All these approaches achieve a formal semantics for the main features of object-oriented programming. Such features, however, are expressed by means of more or less direct encodings in terms of some external formalism, rather than taken as primitive notions. Other approaches, such as the *object calculus* [2], have developed primitive notions to account for objects and message passing. The object calculus can be seen as an extension of the λ-calculus, and it can be combined with rewrite rules to define algebraic operations over data types [13]. Rather than using a λ-calculus, our aim is to develop

an algebraic setting where objects, attributes, methods, classes and inheritance appear as primitive notions. As in [12], we separate the representation of objects as records from the representation of methods as overloaded functions. Following [11]: "(...) a theory of object-oriented programming should first of all focus on the meaning of inheritance", we propose a novel inheritance mechanism that we call *genetic inheritance*, able to model any other particular inheritance behavior, because it is a general form of inheritance, as it is illustrated in the next section.

3. Genetic inheritance and operations

In our approach, objects are grouped by classes, which specify their attributes, or static behavior. Subclasses serve to extend the behavior (new attributes), and/or redefine it (overriding definitions). This setting works assuming multiple attribute inheritance from superclasses to subclasses. Our inheritance mechanism is *genetic* in the sense of the following analogy: classes represent groups of living beings (objects) characterized by some genetic information (existing attributes). Evolution causes subclasses to appear, and genetic information is altered with new features (new attributes), and/or mutated features (attribute overriding). Multiple inheritance with collisions represents a crossover between two possible kinds of values for some feature (attribute); as you would expect it is the living being (the object) which chooses its particular genetic information.

In a previous paper [30] we have modeled this approach within an algebraic framework based on order-sorted signatures. Sorts represent classes, while attributes are specified by selector operations on those sorts, class hierarchies come from the order defined over sorts.

Definition 1. *A GOTA signature* Σ is a triple $(S, \leqslant, A)$, where S is a non-empty finite set of sort symbols, $\leqslant$ is a partial order over S, and A is a (possibly empty) finite set of attribute declarations with symbols of S, such that there are no two proper declarations of the same attribute for any sort.

An attribute declaration $g : s \to w$, where g is an attribute symbol, and s and w are sort symbols, indicates that sort s has attribute g with values in w. We say that such declaration is *proper* for sort s, and it is also a declaration for any sort $s' \leqslant s$, unless there is a proper declaration of g for some sort s'', $s' \leqslant s'' < s$ (attribute overriding and closest inheritance).

Example 2. We have a GOTA signature representing a family

```
sort Smiths.
sorts Blue Green Light.
subsorts Blue Green < Light.
att eyes : Smiths → Light.
```

In examples, sorts are declared by keywords sort, sorts, inheritance (sort order) by subsort, subsorts, and attributes by att. A Smiths family object can be represented with a feature term like Smiths[eyes ⇒ Blue[]]. The notation is similar to that of [2], where objects are described by a record (notation [...]) with their class and attribute values.

When attribute overriding takes place, the lowest existing attribute declaration is the only applicable, based on the sort order. An important concept is that of *complete set of attributes* which corresponds to those possibilities of combining attributes to describe the genetic information of a class (*breeds* of a class).

Definition 3. Given a signature $\Sigma = (S, \leqslant, A)$, and a sort $s \in S$, we define a *complete set of attributes of* s as a collection of attribute declarations $g_j : s_j \to w_j$, with $j = 1, \ldots, m$, such that the g_j are different and all the attributes that s has, and every declaration in the set is a declaration of attribute g_j for s.

Classes which inherit attributes with collisions will have more than one complete set of attributes. We can roughly say that genetic inheritance comes by the hand of complete sets of attributes.

Example 4. We extend the signature for the family

sort Smiths.	sort Jones.
sorts Blue Green Light.	sorts Black Brown Dark.
subsorts Blue Green < Light.	subsorts Black Brown < Dark.
att eyes : Smiths → Light.	att eyes : Jones → Dark.
sorts Small Rounded Human.	sorts Pointed Big Weird.
subsorts Small Rounded < Human.	subsorts Pointed Big < Weird.
att ears : Smiths → Human.	att ears : Jones → Weird.

sort Smith-Jones.
subsort Smith-Jones < Smiths Jones.

Smiths and Jones have just one complete set of attributes, but Smith-Jones have *four* complete set of attributes ({eyes : Smiths → Light, ears : Smiths → Human}, {eyes : Smiths → Light, ears : Jones → Weird}, {eyes : Jones → Dark, ears: Smiths → Human}, {eyes : Jones → Dark, ears : Jones → Weird}), which represent all the combinations of features from the closest superclasses.

A clear advantage of genetic inheritance is that it allows one to solve the *accidental multiple inheritance* problem [38, Chapter 4] which raises when class hierarchies show anomalies (e.g., students that are professors, birds which do not fly, aquatic mammals, etc.). At the same time, this mechanism is able to model any other particular inheritance behavior where attributes names appear once (that is, no qualifying attributes are allowed), since we can form a complete set for every precedence relation. When

propagating this mechanism to methods, we think it can be useful in the design of systems whose objects represent entities that can be in a number of finite modes. Genetic inheritance can be used to separately specify the behavior of every distinct mode as it is shown in the following examples.

3.1. Mutable feature entities as genetic objects

We can imagine many systems whose entities can be in different modes (*on*, *off*, *idle*, *active*, *receiving*, *connecting*, *engaged*, etc.), and those modes are characterized by features with different values, and what is more important, different possible actions. Genetic inheritance can help to model those systems in a more intuitive manner, while providing a seamless way to select appropriate actions.

Example 5. Let us consider a class Chrono of chronometers (devices to measure elapsed time), with an attribute time. We also want a specialized version ChronoSplit able to measure split time. Those chronometers can be set on two modes: measuring split time, or not. In the first case, the additional attribute splitTime must also be updated when time is. But we want to operate in both cases in the same manner (updating can be a physical signal), regardless the chronometer is counting split time, or not. Let us define the following GOTA signature:

```
sort Chrono.
att time : Chrono → Time.
sorts SplitOn SplitOff.
subsorts SplitOn SplitOff < Chrono.
att splitTime : SplitOn → Time.
att splitTime : SplitOff → Void.
sort ChronoSplit.
subsort ChronoSplit < SplitOn SplitOff.
```

Objects of class ChronoSplit are *genetic* in the sense they may inherit attribute splitTime from whatever superclass SplitOn, or SplitOff. These classes represent both modes of split time chronometers. We will define one update operation for both classes, SplitOn and SplitOff. For the first class, time and splitTime are updated (let us say by adding 1), while for the second, only time is updated. The operation can be specified as a function for SplitOn and SplitOff objects:

```
update SplitOn[time ⇒ X, splitTime ⇒ Y] →
            SplitOn[time ⇒ X+1, splitTime ⇒ Y+1]
update SplitOff[time ⇒ X] → SplitOff[time ⇒ X+1]
```

Now, depending on the state of ChronoSplit objects (which inherit from both SplitOn and SplitOff classes), the appropriate definition must be executed, where the state is determined by the sort of attribute splitTime, which can be changed.

The key point in the previous example is that selecting the appropriate action depends on the object mode, which is characterized by the complete set of attributes of the

object. We can view it again in the following example, where genetic inheritance is used to describe a procedure. While the choices of the mutable feature in the previous example were independent, the next example shows an operation that changes from one mode to another, affecting the control of the procedure.

Example 6. A *Russian multiplier* is a device that works as the Russian (or Egyptian) peasant multiplication method. Two natural numbers are multiplied by repeatedly dividing one of them by two, while multiplying the other by two (with some control when dividing odd numbers). We consider the GOTA signature:

```
sorts Mult.
att number: Mult → Nat.
sort RussMultOn.
subsort RussMultOn < Mult.
att times: RussMultOn → Nat.
sort RussMultOff.
subsort RussMultOff < Mult.
att times: RussMultOff → Ok.
sort RussMult.
subsort RussMult < RussMultOff RussMultOn.
```

Objects of class RussMult genetically inherit attribute times from RussMultOn, or RussMultOff. The multiplier is on in the first case (times is Nat), and off in the second (times is Ok). We only require an action for the active mode, which is set when attribute times is updated with a natural number. An operation do proceeds recursively multiplying attribute number by two as long as times is greater than one, which is simultaneously divided by two. When times reaches 1, its value changes to Ok (a constant), and the value of the product is obtained in attribute number. It could be specified as

```
do RussMult[number ⇒ X, times ⇒ 0] →
        RussMult[number ⇒ 0, times ⇒ Ok]
do RussMult[number ⇒ X, times ⇒ 1] →
        RussMult[number ⇒ X, times ⇒ Ok]
do RussMult[number ⇒ X, times ⇒ Y] →
        do RussMult[number ⇒ X*2, times ⇒ Y/2]
    if even(Y) AND Y > 1
do RussMult[number ⇒ X, times ⇒ Y] →
        do RussMult[number ⇒ X*2+X, times ⇒ Y/2]
    if odd(Y) AND Y > 1
```

Actual multipliers are objects of RussMult which may perform do whenever they are set on (when having attribute times of sort Nat, hence inheriting from RussMultOn).

Observe that even though operation do is defined for RussMult objects it is the object mutable feature times which controls the possibility of using the operation. We

may think of using complete sets to characterize the object mode, but, in a more general situation, it can depend on a finer-grain concept. It can be a matter of just one attribute like in the next example.

Example 7. We consider a simplified automatic telling machine (ATM), with two actions: cash withdrawal and accounts balance checking. Operation depends on the state of the ATM: cash availability, and on-line connection. We consider the GOTA signature:

```
sorts NoMoney Money OnLine OffLine.
att money : NoMoney → Void.
att money : Money → Nat.
att line : OffLine → Down.
att line : OnLine → Ok.
sort ATM.
subsort ATM < NoMoney Money OnLine OffLine.
```

Objects of class ATM genetically inherit attributes money, and line. Since every attribute has two possibilities, this amounts up to four complete sets of attributes for ATM. The idea with the operations is that withdraw only works if money is Nat (if there is cash), while balance is only obtained if line is Ok. For a genetic object of ATM it could be two, one, or none of the operations which make sense, depending on the sorts of the values of its attributes.

The last example, aside from motivating the use of genetic inheritance, should have stressed that operation definition depends, not on complete sets of attributes, but on distinguishing values of particular attributes. This will give raise to a typing mechanism more flexible than classes, so-called *genome typing*, which will be introduced in the next section.

A concluding remark about genetic inheritance regards simplicity. We do not claim genetic inheritance to simplify algorithms per se. We think it can help algorithm design in the presence of object orientation. It may not produce a simpler formalization, but there is a more intuitive, declarative view of the problem being modeled. Moreover, using genetic objects allows one to specify a variable behavior for the same method which can produce results in different classes depending on which sort an attribute is. This is not possible using different values of the same sort for an attribute, since the declaration of the method for the sort of the attribute must have a unique codomain.

3.2. Concurrency

The last example should have brought a flavor of interaction. ATM objects may interact with account objects to check the balance. While in our framework, the methods and procedures for this interaction can be specified, since objects act as values for operations, the concurrent setting in which it occurs is out of its scope. This is delib-

erate. We have included as main features the declarative nature of object-orientation: class description via sorts and attributes, and methods and procedures via purely functional operations. We believe that a concurrent setting in which objects interact by message passing can be modeled, *programmed* in the framework itself. The key idea is to view a collection of objects and messages as a meta-object whose evolution can be described by means of appropriate operations. Similar ideas have been already used in other approaches as, e.g., [33].

4. XGOTA signatures

Our first (already achieved in [30]) uniformity argument is that sorts are able to represent classes and data types, treating object and data values as sort elements. More properly, sorts account for classes which are able to describe abstract data types. Now, we will include operations to homogeneously represent methods and class-external functions. In order-sorted approaches, operations are declared by issuing the sorts of the arguments (the domain) and the sort of the result (the codomain) as in $f\colon s_1 \cdots s_n \to s$, where $s_1 \cdots s_n$ is the domain of f, and s the codomain. As we have seen in the previous section, this is not enough to declare the functional type of an operation in GOTA signatures. And this affects not only the domain like in Examples 5–7, but the codomain as well.

Example 8. In the signature of Example 5, we need an operation setSplitOn to set the split time chronometer on (the button that you push in an actual chronometer), which means changing the sort of attribute splitTime. A declaration of setSplitOn must specify that the result has splitTime attribute with values in Times.

4.1. Genome typing

We will use *genetic patterns* to describe the types of attribute values.

Definition 9. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, and a sort $s \in S$, we define a *genetic pattern for* s, G, as a subset of $\mathbf{symb}(A) \times S$, such that: (i) there are not two pairs with the same first attribute symbol, and (ii) for every pair $(g, w) \in G$, there exists a declaration of g for sort s with codomain $w' \geqslant w$.

Notation $\mathbf{symb}(A)$ represents the set of attribute symbols declared in A. Each pair (g, w) (written in examples as g:w) of a genetic pattern G for s, characterizes those objects of s, such that the value of attribute g is of sort w. An ordering can be defined over genetic patterns, which is based on set inclusion, and on the sort order of the signature.

Definition 10. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, we define the *extension of* $\leqslant$ to genetic patterns as follows: $G \leqslant G'$ if and only if for every $(g, w') \in G'$, exists $(g, w) \in G$ such that $w \leqslant w'$, for any genetic patterns G, G'.

Observe that a lower genetic pattern corresponds to more specialized objects, since every attribute is restricted to a lower or equal sort, and conditions over more attributes can be included. If G is lower than G' then the set of objects represented by G is a subset of the set of objects represented by G'. A genetic pattern can be empty, meaning in that case that every object of the corresponding sort is considered. The combination of a sort and a genetic pattern is an object *genome*.

Definition 11. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, we define a *genome* as a pair, $s\{G\}$, where $s \in S$ is a sort symbol, and G is a genetic pattern for s.

With genomes we can declare operations for which we consider a set of operation symbols with an associated arity.

Definition 12. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, an *operation declaration* has the form

$$f : s_1\{G_1\} \cdots s_n\{G_n\} \to s\{G\}$$

where f is an operation symbol, and $s_i\{G_i\}$, $s\{G\}$, are genomes, for $i = 1, \ldots, n$.

Genomes control the declaration of operations in a very powerful way, since we can form the genetic pattern in a wide range of possibilities. As genomes refine sorts, we can introduce genome typing by providing an order over genomes. This is done extending the order of specialization of genetic patterns.

Definition 13. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, we define the *extension of* $\leqslant$ to genomes as follows: $s\{G\} \leqslant s'\{G'\}$ if and only if $s \leqslant s'$, and $G \leqslant G'$, for any genomes $s\{G\}$, and $s'\{G'\}$.

If $s\{G\}$ is lower than $s'\{G'\}$ then the set of objects represented by $s\{G\}$ is a subset of the set of objects represented by $s'\{G'\}$.

4.2. Multiple operation declaration

As subclasses extend and modify the attributes with respect to superclasses, the meaning of operations can be refined while descending the class hierarchy. The idea is to *override* any previous meaning of a method for the objects of a subclass where the method has been redefined. The syntactic way to support overriding in order-sorted signatures is to allow *multiple operation declarations*, which establish different domains, codomains and meanings for the same operation. In order to determine which of the available declarations must be applied to a particular object we use the specialization order over genomes given by Definition 13.

We extend the order between genomes to *tuples* of genomes considering every genome in the tuple. When this condition occurs for the tuple of arguments in the

domain of an operation declaration we can say that the declaration with more specialized genomes in the domain *overrides* the other one.

Definition 14. Given a GOTA signature $\Sigma=(S,\leqslant,A)$, and two declarations for the same operation f

$$f: s_1\{G_1\}\cdots s_n\{G_n\}\to s\{G\} \quad f: s'_1\{G'_1\}\cdots s'_n\{G'_n\}\to s'\{G'\}$$

we say that the first declaration *overrides* the second one if and only if

$$(s_1\{G_1\}\cdots s_n\{G_n\}) < (s'_1\{G'_1\}\cdots s'_n\{G'_n\})$$

Note that overriding of declarations is strict. We can consider which declarations of a given operation f are *applicable* to a tuple of genomes just by comparing it with the declaration domain.

Definition 15. Given a GOTA signature $\Sigma=(S,\leqslant,A)$, we say that a declaration of an operation f with arity n, $f: w_1\{G_1\}\cdots w_n\{G_n\}\to w\{G\}$, is *applicable* to a given n-tuple of genomes $w'_1\{G'_1\}\cdots w'_n\{G'_n\}$ if and only if $(w'_1\{G'_1\}\cdots w'_n\{G'_n\})\leqslant(w_1\{G_1\}\cdots w_n\{G_n\})$.

We are interested in *minimal declarations*.

Definition 16. Given a GOTA signature $\Sigma=(S,\leqslant,A)$, and a set F of operation declarations, we say that a declaration of an operation f with arity n, is *minimal* for a given n-tuple of genomes $w_1\{G_1\}\cdots w_n\{G_n\}$ if and only if such declaration is applicable, and it is not overridden by another declaration of f in F, applicable to $w_1\{G_1\}\cdots w_n\{G_n\}$.

Observe that multiple declarations may also overload operations for different, not related by hierarchy, sorts.

4.3. Auto-declarations and mutator methods

Inheritance of operations poses a problem for declarations. Specifically, we want to declare an operation such that the class of the result is the same as that of the argument, but we also want that to hold when the operation is inherited in subclasses. This is the situation when we have a method which modifies the object is aimed to, a *mutator method.*

Example 17. In the Russian multipliers signature of Example 6, we need an operation setNumber to modify the number attribute of class Mult. Its declaration could be (in examples we will use op to represent operation declarations)

op setNumber: Mult{ } → Mult{ }.

which is applicable by inheritance in subclass RussMult. But with this "type", any expression constructed with setNumber will be of genome Mult{ }, even though the argument would be a RussMult.

This is part of the problem order-sorted signatures have when encoding object-oriented characteristics. We need a kind of *Self* type for these operations which can be used to obtain the type of expressions. Nevertheless, with genetic inheritance, what we want to preserve is only the sort, since the genetic pattern might change in the case it represents a different mutable feature value for a genetic object.

Example 18. For the signature of Example 5, we try to declare operation setSplitOn to set the split time chronometer on. Remember that this implies changing the sort of the attribute splitTime from Void to Time. Using a self type for the declaration of setSplitOn it could be

op setSplitOn : ChronoSplit{splitTime : Void} → Self.

In this case, expressions with setSplitOn would correctly have the sort of any given subclass, but would maintain genetic pattern splitTime:Void.

The idea is to explicitly represent the changes in the genetic pattern of the result. We will use *auto-operation declarations* which combine self type with genetic pattern modification.

Definition 19. Given a GOTA *signature* $\Sigma = (S, \leqslant, A)$, an *auto-operation declaration* has the form

$$f : s_1\{G_1\} \cdots s_n\{G_n\} \to auto\{G\}$$

where f is an operation symbol, s_i $\{G_i\}$, are genomes, for $i = 1, \ldots, n$, and G is a genetic pattern for s_1.

The notation $auto\{G\}$ should be understood as *auto* (meaning the genome of *self*) modified by G. We intend to associate the effect of auto-declarations to the first argument genome, assuming that these declarations correspond to mutator methods, where the first argument plays the role of the receiving object. Observe that G must be a genetic pattern for s_1, that is, we only allow changes that would fit in any actual first argument. What about the genome of the result? We will use the following transformation of genetic patterns.

Definition 20. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, and two genetic patterns G, and G' for a sort s, we define the *transformation of* G *by* G', noted as $G \leftarrow G'$, as the genetic pattern for s consisting of pairs (g, w), such that: $(g, w) \in G'$, or otherwise, $(g, w) \in G$, and g is not in any pair of G'.

Observe that G' takes precedence in the transformation, imposing its pairs in the case of attribute coincidence. Now, for any first argument with genome $s\{G\}$, if G' represents the effect of some method f in the genetic pattern of the first argument, the result would fulfill genome $s\{G \leftarrow G'\}$. However, that may fail to be a genome if s has redefined the attributes appearing in G', that is, if s no longer has the attributes in G' defined as in G'. This point restricts the applicability of auto-declarations, and the model of inheritance for this kind of operations.

Definition 21. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, we say that an auto-declaration of an operation f with arity n, $f : w_1\{G_1\} \cdots w_n\{G_n\} \rightarrow auto\{G\}$, is *applicable* to a given n-tuple of genomes $w'_1\{G'_1\} \cdots w'_n\{G'_n\}$ if and only if $(w'_1\{G'_1\} \cdots w'_n\{G'_n\}) \leqslant (w_1\{G_1\} \cdots w_n\{G_n\})$, and G is a genetic pattern for w'_1.

The restriction of G being genetic pattern for w'_1, corresponds to the idea that what we are doing with a mutator method is modifying values in some object attributes. Since the new values correspond to genetic pattern G, sort w'_1 of the actual first argument must admit those values. Observe that this is not guaranteed simply by having $w'_1 \leqslant w_1$ because, w'_1 can have some attributes appearing in G redefined, not accepting values as those of G.

Example 22. Now, for the Russian multipliers example, we can declare operation setNumber as

op setNumber : Mult{ } → auto{ }.

And for the chronometers example, operation setSplitOn can be

op setSplitOn : SplitOff{splitTime : Void} → auto{splitTime:Time}.

In both cases, expressions with these operations can now be well-genome typed.

Observe that this mechanism is able to represent those particular situations when an operation f does not affect the genome of the object receiving the method. It suffices to declare f as

$$f : w_1\{G_1\} \cdots w_n\{G_n\} \rightarrow auto\{\}$$

which means that it would be applicable to any first argument fulfilling genome $s\{G\} \leqslant w_1\{G_1\}$ – since the condition of $\{\}$ being genetic pattern for s trivially holds –, and the result would fulfill genome $s\{G \leftarrow\}$, which is $s\{G\}$. Note that an approach which only permits this kind of mutator methods would never be able to declare operations to change mutable features of a genetic object.

4.4. Regularity

Though being a necessary mechanism, multiple declarations of operations must be regulated. This means that there cannot be contradictory declarations, in the sense of

two (or more) declarations specifying *different actions* for the *same object*. This is not a normal situation, but for class hierarchies where multiple inheritance is allowed it can naturally occur. Non-ambiguity is important when evaluating expressions, to properly perform reductions, and we need to assure that for any object at most one declaration of any operation (and its associated definition rules) is applicable. This corresponds to a *regularity property* of the set of operation declarations.

Definition 23. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, and a set F of operation declarations, we say that F is *regular* if and only if, for every operation symbol f of arity n, there exists at most one minimal declaration of f for any n-tuple of genomes $w_1\{G_1\} \cdots w_n\{G_n\}$.

This condition should not be seen as a limitation for operation declaration, but as a regulation which allows to know the intended "shape" of objects which can perform an operation (in the case of methods), or objects that can be subjected to an operation (in the case of external procedures). In many situations, assuring regularity is a matter of good programming style in the presence of genetic inheritance.

In contrast to [12], our regularity condition does not impose any covariance requirements. When an operation declaration is overridden by another one which applies to objects with more specialized genomes, we do not require a more specialized genome for the result. As we will see, this more liberal overriding policy precludes dynamic dispatch. Nevertheless, we can ensure static well typing. Moreover, our semantics offers an undefined value to account for situations where no applicable operation declaration exists. A formal treatment of these issues is given in Sections 6 and 7.

4.5. Extending GOTA signatures with operations

XGOTA signatures will be defined from three disjoint sets of symbols, one for the *sorts*, one for the *attributes*, and one for the *operations*, which have a fixed arity. The idea is augmenting a GOTA signature with a regular set of operation declarations.

Definition 24. A *GOTA signature extended with operations*, or *XGOTA signature*, for short, Σ is a tuple $(S, \leqslant, A, F)$, where $(S, \leqslant, A)$ is a GOTA signature, and F is a finite regular set of operation declarations over the sorts of S, and the attributes of A.

Note that in an XGOTA signature the set of sorts S, and the set of attribute declarations A, are finite, since this is required for $(S, \leqslant, A)$ to be a GOTA signature. This makes regularity decidable for a finite set of operation declarations. As we were seeking, operations are a homogeneous mechanism to specify methods and external-class functions. This is possible given that sorts represent classes as well as data types, making for declaration uniformity. Additionally, when dealing with purely functional objects, methods and external-class functions do not differ except in encapsulation. In our approach that is an implementation issue which can be easily achieved just by not allowing object patterns (which makes for object representation access) to be used

in the left-hand side of defining rewriting rules for functions. Lastly, the problem of mutator method inheritance is solved by the inclusion of auto-declarations as we have explained before.

An important point to understand is that we want to make a distinction between abstract objects and the effect of applying operations to them. In GOTA signatures we had feature terms to represent objects (a static behavior description), and in XGOTA signatures we will add *expressions* to represent the effects of actions (a dynamic behavior description).

Given an XGOTA signature $\Sigma = (S, \leqslant, A, F)$, we will have a set V of variables, such that every variable has an *associated genome*, and V has infinitely many variables of every genome. Variables will be denoted by letters x, y, z, the notation $\mathbf{mg}(x)$ will denote the associated genome of a variable x, and we will refer to subsets of variables belonging to V, denoted with capital letters X, Y, Z. Expressions will use genome typing, which is needed to define them.

Definition 25. Given an XGOTA signature $\Sigma = (S, \leqslant, A, F)$, the set of *$\Sigma$-expressions with genome $s\{G\}$* is inductively defined as follows:

- (Variables). $x \in V$, if $\mathbf{mg}(x) \leqslant s\ \{G\}$.
- (Feature expressions). $w[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]$, if $w \in S$, the attribute symbols g_j are different, there exist declarations for w, $g_j : w_j \rightarrow s_j \in A$, every e_j is a Σ-expression with genome $v_j\{G_j\} \leqslant s_j\{\}$, for $j = 1, \ldots, m$, and $w\{(g_1, v_1), \ldots, (g_n, v_n)\} \leqslant s\{G\}$.
- (Tagged expressions). $x : w[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]$, if $\mathbf{mg}(x) \leqslant s\ \{G\}$, and $w[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]$ is a Σ-expression with genome $\mathbf{mg}(x)$.
- $f(e_1, \ldots, e_n)$, if every e_i is a Σ-expression, where $w_i\{G_i\}$ is the minimum genome of e_i, for $i = 1, \ldots, n$, and there exists a minimal declaration d of f for $w_1\{G_1\} \cdots w_n\{G_n\}$, such that:
  - if d is $f : w'_1\{G'_1\} \cdots w'_n\{G'_n\} \rightarrow w'\{G'\}$, then $w'\{G'\} \leqslant s\{G\}$, and
  - if d is $f : w'_1\{G'_1\} \cdots w'_n\{G'_n\} \rightarrow auto\{G'\}$, then $w_1\{G_1 \leftarrow G'\} \leqslant s\{G\}$.

As genomes can be considered as "typing constraints" we will also say that an expression with genome $s\{G\}$, fulfills that genome. Observe that in the fourth point of the previous definition, in the case of the auto-declaration, since d is minimal, it is applicable, and G' is a genetic pattern for w_1, which implies that the transformation $w_1\{G_1 \leftarrow G'\}$ is correctly defined by Definition 20. The previous definition depends on having a minimum genome for expressions, which is possible owing to the election of the minimal declaration in the fourth point. The concept of class inclusion is applied to expressions by the partial order defined over genomes, accepting as elements with genome $s'\{G'\}$ also those elements with a more specialized genome $s\{G\} \leqslant s'\{G'\}$.

Tagged expressions [31] (an idea borrowed from the ψ-terms of [6]) allow one to define recursive objects, making it possible to express relations between objects and their attributes. They also serve to build infinite data structures in combination with operations as we will see later on.

Example 26. We define a signature for people:

sorts Person.
att loves: Person → Person.
att spouse: Person → Person.
vars X Y: Person{ }.

(In examples, variables are capital letters declared using keywords var, or vars.) We can form the following tagged expression:

X:Person[spouse ⇒ Y:Person[spouse ⇒ X]]

to represent the one-to-one marriage relationship. And

X:Person[loves ⇒ Y:Person[loves ⇒ X, spouse ⇒ X], spouse ⇒ Y]

to represent the desirable relationship of love and marriage.

The set of variables that occur in an expression is relevant for various concepts such as ground expressions, substitutions, or assignments. Nevertheless, since variables may act as tags in an expression, we have to distinguish *free occurrences* of variables. Like quantifiers, when we issue a tag we are binding the tag variable to some expression (to its value), and the scope of the tag is the corresponding tagged expression. We will say that an occurrence of a variable x is *bound* if it is inside of an x-tagged expression; otherwise, we will say that it is *free*. We define the sets of *variables* and *free variables* of an expression.

Definition 27. Given an XGOTA signature $\Sigma=(S,\leqslant,A,F)$, and a Σ-expression e, the sets of *variables*, and free *variables* of e, denoted by $\mathbf{var}(e)$, and $\mathbf{fvar}(e)$, respectively, are inductively defined as follows:

- If $e\equiv x,\ x\in V$ then

$$\mathbf{var}(e)=_{\mathrm{def}}\{x\},\quad \mathbf{fvar}(e)=_{\mathrm{def}}\{x\}.$$

- If $e\equiv w[g_1\Rightarrow e_1,\ldots,g_m\Rightarrow e_m]$ then

$$\mathbf{var}(e)=_{\mathrm{def}}\bigcup_{j=1}^{m}\mathbf{var}(e_j),\quad \mathbf{fvar}(e)=_{\mathrm{def}}\bigcup_{j=1}^{m}\mathbf{fvar}(e_j).$$

- If $e\equiv x:w[g_1\Rightarrow e_1,\ldots,g_m\Rightarrow e_m]$ then

$$\mathbf{var}(e)=_{\mathrm{def}}\{x\}\cup\mathbf{var}(w[g_1\Rightarrow e_1,\ldots,g_m\Rightarrow e_m]),$$

$$\mathbf{fvar}(e)=_{\mathrm{def}}\mathbf{fvar}(w[g_1\Rightarrow e_1,\ldots,g_m\Rightarrow e_m])\backslash\{x\}.$$

- If $e\equiv f(e_1,\ldots,e_n)$ then

$$\mathbf{var}(e)=_{\mathrm{def}}\bigcup_{i=1}^{n}\mathbf{var}(e_i),\quad \mathbf{fvar}(e)=_{\mathrm{def}}\bigcup_{i=1}^{n}\mathbf{fvar}(e_i).$$

An expression e is *ground* if no free variable occurs in it. The set of Σ-expressions formed with free variables of a set X is denoted by $E_\Sigma(X)$, more precisely, we define $e \in E_\Sigma(X)$ if and only if e is a Σ-expression, and $\mathbf{fvar}(e) \subseteq X$. The set of ground Σ-expressions is denoted by $E_\Sigma(\emptyset)$, or just E_Σ. We will also use the notation $E_{\Sigma,s\{G\}}(X)$ to refer to the set of expressions with genome $s\{G\}$ formed with free variables of $X \subseteq V$.

We will find interesting to recognize those expressions which do not include any operation symbol, they will be *terms*. Given a signature Σ, the set of Σ-terms formed with free variables of a set X will be denoted by $T_\Sigma(X)$. The set of ground Σ-terms will be denoted by T_Σ. We will also say that a term t is *simple* if it does not include tags. Terms will be used for defining operations, for which we will need a little bit more of notation. We would want to distinguish the attribute description part of a feature expression.

Definition 28. Given an extended GOTA signature $\Sigma = (S, \leqslant, A, F)$, we define a *$\Sigma$-description* with genetic pattern G as $g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m$, where the attribute symbols g_j are different, there exist declarations $g_j : w_j \to s_j \in A$ for some sort w, every e_j is a Σ-expression with genome $v_j\{G_j\} \leqslant s_j\{\}$, for $j = 1, \ldots, m$, and $\{(g_1, v_1) \ldots (g_m, v_m)\} \leqslant G$. We also define

$$\mathbf{var}(g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m) =_{\text{def}} \bigcup_{j=1}^{m} \mathbf{var}(e_j),$$

$$\mathbf{fvar}(g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m) =_{\text{def}} \bigcup_{j=1}^{m} \mathbf{fvar}(e_j).$$

With descriptions we can ease the notation for feature expressions, highlighting the existence of a particular attribute and its value. If ld represents the description $g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m$, and $g \not\equiv g_j$, for $j = 1, \ldots, m$, we will use the notation $w[g \Rightarrow e \mid ld]$, to represent the feature expression whose description is formed by $g_j \Rightarrow e_j$, for $j = 1, \ldots, m$, and $g \Rightarrow e$. It is important to remark that, even though the order of attributes in a description makes syntactically different expressions, they represent the same object.

Class inclusion is also applied to descriptions by the partial order defined over genetic patterns, accepting as descriptions with genetic pattern G', those descriptions with genetic pattern G, as long as $G \leqslant G'$.

4.6. Substitutions

In XGOTA signatures, variables have an associated (minimum) genome, and when we define substitutions we must preserve genomes.

Definition 29. Given an XGOTA signature $\Sigma = (S, \leqslant, A, F)$, we define a *$\Sigma$-substitution* as a mapping $\theta : V \to E_\Sigma(V)$ such that if $x \in V$, then $\theta(x)$ fulfills genome $\mathbf{mg}(x)$.

This definition makes possible to preserve genomes, that is, if x fulfills $s\ \{G\}$ then $\theta(x)$ fulfills $s\ \{G\}$. Note that $\theta(x)$ is an expression with any genome that x fulfills, but $\theta(x)$ may fulfill a more specialized genome. We will use the usual postfix notation for applying substitutions. In the case of a substitution θ having a finite domain, θ can be represented by a *set of links* $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where t_i is $x_i\theta$, for $i = 1, \ldots, n$. We can also make explicit the domain and range of a substitution with the notation: $\theta : X \to E_\Sigma(Y)$ which means that $\mathbf{dom}(\theta) \subseteq X \subseteq V$, and $\mathbf{ran}(\theta) \subseteq Y \subseteq V$.

We can apply substitutions to expressions replacing every variable in an expression. But we must realize that variables in expressions may be bound with tags, and only free occurrences of variables can be replaced. We must also avoid introducing variables that may become bound by an existing tag.

Example 30. Using the signature of Example 26, if we consider an expression

X:Person[loves ⇒ X, spouse ⇒ Y]

replacing Y with X will produce

X:Person[loves ⇒ X, spouse ⇒ X]

where the value of attribute spouse has been captured by tag X.

This *variable capturing* problem is solved simply by renaming existing tags if needed when applying substitutions. In the previous example, renaming X by Z makes the expression Z:Person[loves ⇒ Z, spouse ⇒ Y] safe for replacing Y with X.

Expressions obtained when applying substitutions are *instances* of the original expressions. Instances of a given expression e are more specific expressions, since they maintain the structure of e, while changing some variables by expressions that can be more detailed. This is particularly true for the minimum genome, which can "become specialized" in instances. May this behavior produce ill-formed expressions as a result of applying a substitution? It could be because of the definition of expressions with operations, where the minimal declaration of a given operation is used. As substitutions "descend" over the genomes order, we could obtain different minimum genomes.

Example 31. In the following, XGOTA signature we portrait a particular case of animal life:

```
sorts  Birds  Penguins.
subsort  Penguins < Birds.
att  flight:Birds → Nat.
att  flight:Penguins → Bool.
op  speed:Birds{ } → Nat{ }.
op  speed:Penguins{ } → Bool{ }.
var  B:Birds{ }.
```

where Nat and Bool are already specified sorts of natural numbers and boolean values, respectively. Let us assume there exists an operation double to duplicate a natural number. We consider expression

double(speed(B))

with minimum genome Nat{ }, since the minimal declaration of speed for the genome of B, Birds{ } is applied. Now, we consider a substitution μ such that

$$\mu(\mathsf{B}) = \mathsf{Penguins}[\,]$$

which is correctly defined, since expression Penguins[ ] fulfills the associated genome of B, Birds{ }. If we apply μ to double (speed(B)), we get

double(speed(Penguins[]))

which is an ill-formed expression since speed(Penguins[]) has minimum genome Bool{ }, not being a correct argument for operation double.

The possibility of non-monotonically redefining operations while specifying subclasses may cause substituted expressions to fail to preserve genomes, and to be well formed. Fortunately, this problem does not refer to terms, which behave adequately since they do not include operation symbols, and only well-formed expressions are obtained when applying a substitution to a term.

5. XGOTA algebras

To obtain the meaning of operations, our idea is to use continuous applications which allow us to model lazy evaluation with potentially infinite objects. In order to deal with inheritance and (possibly non-covariant) overriding, we are going to overload operation denotations with one mapping for every declaration. This way, we try to guarantee that the declaration used to construct an expression is the one used to compute its denotation, as well. This will induce the static typing later on.

We will use some domain theoretic definitions and results [22]. A *partial order* (poset) is any set P with a partial order $\sqsubseteq$ defined over it. Given a poset $(P, \sqsubseteq)$, and a subset $Q \subseteq P$, we say that p is an *upper bound* of Q if for all $q \in Q$, $q \sqsubseteq p$. We say that p is the *least upper bound* of Q if p is an upper bound of Q, and for all upper bound q of Q, $p \sqsubseteq q$. We will denote the least upper bound of a set Q with $\bigsqcup Q$. Given a poset $(P, \sqsubseteq)$, we say that a subset $X \subseteq P$ is *directed* if every finite subset of X has an upper bound in X. This is equivalent to requiring that for every $x, y \in X$, there exists $z \in X$ such that $x \sqsubseteq z$, and $y \sqsubseteq z$. A *complete partial order* (CPO) is a triple $(P, \sqsubseteq, \perp)$, where $(P, \sqsubseteq)$ is a poset, $\perp \in P$ is a least element, called *bottom*, and every directed set $Q \subseteq P$ has a least upper bound, $\bigsqcup Q \in P$.

Given a CPO $(P, \sqsubseteq, \perp)$, we say that $p \in P$ is a *finite element* if whenever $p \sqsubseteq \bigsqcup Q$, for a non-empty directed set Q, then there exists $q \in Q$, such that $p \sqsubseteq q$. We say that

$p \in P$ is *total* whenever it is maximal with respect to $\sqsubseteq$. We will say that p is *totally defined*, if it is finite and total. The set of all totally defined elements of P will be denoted by $\mathbf{tot}(P)$. Given a CPO $(P, \sqsubseteq, \bot)$, we say that it is *algebraic* if every $p \in P$ is the least upper bound of a directed set of finite elements.

Given two CPOs $(P_1, \sqsubseteq_1, \bot_1)$, and $(P_2, \sqsubseteq_2, \bot_2)$, we say that a mapping $f : P_1 \to P_2$ is *continuous* if it is monotonic, and for every directed set $\{x_i\}_{i \in I} \subseteq P_1$ it holds

$$f\left(\bigsqcup_{i \in I} \{x_i\}\right) = \bigsqcup_{i \in I} f(x_i).$$

Given a partial order $(P, \sqsubseteq)$, an *increasing chain* is a sequence $p_0 \sqsubseteq p_1 \sqsubseteq \cdots \sqsubseteq p_n \sqsubseteq \cdots$ of elements $p_i \in P$, for $i = 1, 2, \ldots, n, \ldots$.

In our semantics we want to abstract away the order of attributes. Therefore, we assume the existence of a total ordering defined over the set of attribute symbols of any XGOTA signature. This makes possible to speak about *ordered* (complete) sets of attributes.

Definition 32. Given an XGOTA signature $\Sigma = (S, \leqslant, A, F)$, an *XGOTA Σ-algebra* $\mathscr{A}$ consists of *denotations* for the sort symbols $s^{\mathscr{A}}$, the attribute symbols $g^{\mathscr{A}}$ of Σ, *semantic constructors* $c_s^{\mathscr{A}}$ for any $s \in S$, and *denotations* $f_d^{\mathscr{A}}$ for every declaration d of the operation symbols, such that the following conditions hold:

(1) $\mathscr{C}_{\mathscr{A}}$ the *carrier set* of $\mathscr{A}$ is a CPO $(\mathscr{C}_{\mathscr{A}}, \sqsubseteq, \bot)$.

(2) For every genome $s\{G\}$, $(s\{G\})^{\mathscr{A}} \subseteq \mathscr{C}_{\mathscr{A}}$ is a CPO $((s\{G\})^{\mathscr{A}}, \sqsubseteq, \bot)$ (with the restriction of $\sqsubseteq$), such that if $s\{G\} \leqslant s'\{G'\}$ then $(s\{G\})^{\mathscr{A}} \subseteq (s'\{G'\})^{\mathscr{A}}$. For every sort $s \in S$, $s^{\mathscr{A}}$ is $(s\{\})^{\mathscr{A}}$.

(3) For every sort s with n attributes $(n \geqslant 0)$, $c_s^{\mathscr{A}} : (\mathscr{C}_{\mathscr{A}})^n \to \mathscr{C}_{\mathscr{A}}$ is a continuous mapping such that, for every $a_1 \in \mathscr{C}_{\mathscr{A}}, \ldots, a_n \in \mathscr{C}_{\mathscr{A}}$,
   - if there exists an ordered complete set of attributes of s, $g_i : s_i \to w_i \in A$ with $a_i \in w_i^{\mathscr{A}}$, for $i = 1, \ldots, n$, then $c_s^{\mathscr{A}}(a_1, \ldots, a_n)$ is a value of $s^{\mathscr{A}}$, finite whenever the a_i are, and total whenever the a_i are,
   - $c_s^{\mathscr{A}}(a_1, \ldots, a_n)$ is $\bot$, otherwise.

(4) For every declaration $d, f : s_1\{G_1\} \cdots s_n\{G_n\} \to s\{G\} \in F$, $f_d^{\mathscr{A}} : (s_1\{G_1\})^{\mathscr{A}} \cdots (s_n\{G_n\})^{\mathscr{A}} \to (s\{G\})^{\mathscr{A}}$ is a continuous mapping.

(5) For every auto declaration $d, f : s_1\{G_1\} \cdots s_n\{G_n\} \to auto\{G\} \in F$, $f_d^{\mathscr{A}} : (s_1\{G_1\})^{\mathscr{A}} \cdots (s_n\{G_n\})^{\mathscr{A}} \to (s_1\{G_1 \leftarrow G\})^{\mathscr{A}}$ is a continuous mapping, such that for every $a_1 \in (s_1\{G_1\})^{\mathscr{A}}, \ldots, a_n \in (s_n\{G_n\})^{\mathscr{A}}$, if $a_1 \in (s_1'\{G_1'\})^{\mathscr{A}}$, $s_1'\{G_1'\} \leqslant s_1\{G_1\}$, and G is genetic pattern for s_1', then $f_d^{\mathscr{A}}(a_1, \ldots, a_n) \in (s_1'\{G_1' \leftarrow G\})^{\mathscr{A}}$.

Fourth and fifth conditions guarantee that every operation declaration is denoted with a distinct mapping. Besides, those mappings are restricted to the appropriate values by considering the genome denotations corresponding to the declaration. Genome denotations are defined by condition (2) as the set of values that fulfill a given genome, which must be a CPO with the restriction of the order $\sqsubseteq$. Observe that $\bot$ is a value that

fulfills every genome. Note also that as sort s is genome $s\{\}$, it also holds $s^{\mathscr{A}} \subseteq w^{\mathscr{A}}$ when $s \leqslant w$ because $s\{\} \leqslant w\{\}$, and $(s\{G\})^{\mathscr{A}} \leqslant (s\{\})^{\mathscr{A}}$ since $s\{G\} \leqslant s\{\}$.

5.1. The meaning of expressions

Variables occurring in expressions obtain a meaning when they are assigned an abstract value of their associated genome.

Definition 33. Given an XGOTA signature Σ, and an XGOTA Σ-algebra $\mathscr{A}$, a variables assignment in $\mathscr{A}$ for the set of variables $X \subseteq V$, also called *$\mathscr{A}$-assignment for* X, is defined as any mapping $\alpha : X \to \mathscr{C}_{\mathscr{A}}$, such that, for any variable $x \in X$, $\alpha(x) \in (\mathbf{mg}(x))^{\mathscr{A}}$. We will say that α is *totally defined* if $\alpha(x) \in \mathbf{tot}(\mathscr{C}_{\mathscr{A}})$, for any variable $x \in X$.

Definition 34. Given an extended signature Σ, an extended continuous Σ-algebra $\mathscr{A}$, and a variables assignment α for the variables of a set $X \subseteq V$, we define an *extension of* α for $z \in X$, $\alpha[z/a]$, with $a \in (\mathbf{mg}(z))^{\mathscr{A}}$, as

$$(\alpha[z/a])(x) =_{\text{def}} \begin{cases} \alpha(x) & \text{if } x \not\equiv z, \\ a & \text{if } x \equiv z \end{cases}$$

for every variable $x \in X$.

We will define the denotation of a tagged expression with approximations, as the *least fixpoint* of certain operator. We have the following known result for the least fixpoint of a continuous operator [41].

Proposition 35. *Let $(A, \sqsubseteq, \bot)$ be a CPO, for every continuous operator $\Phi : A \to A$ there exists the least fixpoint of Φ, $\mathbf{fix}(\Phi)$, which is the least value $a \in A$ such that $\Phi(a) = a$. Moreover, $\mathbf{fix}(\Phi) = \bigsqcup_{n \in \mathbb{N}} \Phi \uparrow n$, where $\Phi \uparrow 0 = \bot$, and $\Phi \uparrow n + 1 = \Phi(\Phi \uparrow n)$.*

Now, we can define the *denotation of expressions*.

Definition 36. Given an XGOTA signature Σ, an XGOTA Σ-algebra $\mathscr{A}$, an expression e of $E_\Sigma(X)$, and an $\mathscr{A}$-assignment α for X, the *denotation of e in $\mathscr{A}$ under α*, $[\![e]\!]_\alpha^{\mathscr{A}}$ is inductively defined as follows:

- If $e \equiv x$, $x \in X$ then $[\![e]\!]_\alpha^{\mathscr{A}} =_{\text{def}} \alpha(x)$.
- If $e \equiv w[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]$, where $g'_1, \ldots, g'_n$ is the ordered set of attributes of w then $[\![e]\!]_\alpha^{\mathscr{A}} =_{\text{def}} c_w^{\mathscr{A}}(a_1, \ldots, a_n)$, where

$$a_i = \begin{cases} \bot & \text{if } g'_i \neq g_j \text{ for any } j = 1, \ldots, m, \\ [\![e_j]\!]_\alpha^{\mathscr{A}} & \text{if } g'_i = g_j \text{ for some } j \in \{1, \ldots, m\} \end{cases}$$

for $i = 1, \ldots, n$.

- If $e \equiv x : e'$ then $[\![e]\!]^{\mathscr{A}}_{\alpha} =_{\text{def}} \mathbf{fix}(\Phi)$, where $\Phi : (\mathbf{mg}(x))^{\mathscr{A}} \to (\mathbf{mg}(x))^{\mathscr{A}}$ is the operator $\Phi(b) =_{\text{def}} [\![e']\!]^{\mathscr{A}}_{\alpha[x/b]}$.
- If $e \equiv f(e_1, \ldots, e_n)$ then $[\![e]\!]^{\mathscr{A}}_{\alpha} =_{\text{def}} f^{\mathscr{A}}_{d}([\![e_1]\!]^{\mathscr{A}}_{\alpha}, \ldots, [\![e_n]\!]^{\mathscr{A}}_{\alpha})$, where d is the minimal declaration for the tuple $\mathbf{mg}(e_1) \cdots \mathbf{mg}(e_n)$.

Observe that in the second point, $\{g_1, \ldots, g_m\} \subseteq \{g'_1, \ldots, g'_n\}$ holds by Definition 25 since $e \in E_{\Sigma}(X)$. Note also that the last item in the definition selects the interpretation of an overloaded operation, according to the minimal declaration which can be applied to the minimal genomes of the arguments. Finally, observe that the third item relies on the least fixpoint of an operator Φ to obtain the denotation of a tagged expression. This is correct, because Φ is a continuous operator, as shown in the next proposition.

Proposition 37. *Let Σ be an extended signature, $e \in E_{\Sigma}(X)$ an expression, and $\mathscr{A}$ an extended continuous Σ-algebra. For every $\mathscr{A}$-assignment α for X, it holds that if e fulfills genome $s\{G\}$ then $[\![e]\!]^{\mathscr{A}}_{\alpha}$ is well defined, $[\![e]\!]^{\mathscr{A}}_{\alpha} \in (s\{G\})^{\mathscr{A}}$, and the operator $\Phi : (\mathbf{mg}(z))^{\mathscr{A}} \to (\mathbf{mg}(z))^{\mathscr{A}}$, with $\Phi(b) = [\![e]\!]^{\mathscr{A}}_{\alpha[z/b]}$, is defined and continuous for $z \in X$, such that $\mathbf{mg}(e) \leqslant \mathbf{mg}(z)$.*

Proof (*Sketch*). It can be shown by structural induction over Σ-expression e, that $[\![e]\!]^{\mathscr{A}}_{\alpha}$ is well defined, $[\![e]\!]^{\mathscr{A}}_{\alpha} \in (s\{G\})^{\mathscr{A}}$, and for any directed set $\{b_k\}_{k \in K} \subseteq (\mathbf{mg}(z))^{\mathscr{A}}$ it holds

$$\bigsqcup_{k \in K} [\![e]\!]^{\mathscr{A}}_{\alpha[z/b_k]} = [\![e]\!]^{\mathscr{A}}_{\alpha[z/\bigsqcup_{k \in K} b_k]} \tag{1}$$

which will allow us to show that Φ is defined – because the premises consider an arbitrary assignment – and continuous for $z \in X$, since $((\mathbf{mg}(z))^{\mathscr{A}}, \sqsubseteq, \bot)$ is a CPO by Definition 32 (2.), and

$$\bigsqcup_{k \in K} \Phi(b_k) = \bigsqcup_{k \in K} [\![e]\!]^{\mathscr{A}}_{\alpha[z/b_k]} =_1 [\![e]\!]^{\mathscr{A}}_{\alpha[z/\bigsqcup_{k \in K} b_k]} = \Phi\left(\bigsqcup_{k \in K} b_k\right),$$

where $(=_1)$ holds by Eq. (1). □

6. Operation definition and rewrite calculus

XGOTA signatures allow us to introduce operation declarations for object hierarchies. But declarations only describe the domain and codomain, given by genomes, and we also want to specify the effect of a given operation. To do this we will use *rewrite rules*.

Definition 38. Given an XGOTA signature $\Sigma = (S, \leqslant, A, F)$, we define a *$\Sigma$-rewrite rule* with label $d \in F$, written as

$$d : f(t_1, \ldots, t_n) \to r \Leftarrow C$$

where d is a declaration, $f : s_1\{G_1\}, \ldots, s_n\{G_n\} \to s\{G\}$, $r \in E_{\Sigma}(V)$ with genome $s\{G\}$, $t_i \in T_{\Sigma}(V)$ is a simple term with genome $s_i\ \{G_i\}$, for $i = 1, \ldots, n$, $f(t_1, \ldots, t_n)$ is

linear – there are not two occurrences of the same variable – $\mathbf{fvar}(r) \subseteq \mathbf{fvar}(f(t_1,\ldots,t_n))$, and C is a finite set (possibly empty) of conditions of the form $e == e'$, where $e, e' \in E_\Sigma(V)$.

Conditions are conjunctions of strict equalities $e == e'$, representing that both expressions e, and e' can be reduced to the same totally defined (computable) value. The need of labeling rules arises if we think of multiple declarations for the same operation for which the pattern of application is the same. We need to distinguish which rules correspond to which declarations. Linearity in left-hand sides is necessary to guarantee that operation declarations can be denoted with continuous mappings.

However, auto-operations (used to represent mutator methods) are special in the sense they can access and modify the attributes of the object given as first argument. Normally, this means not changing every attribute of the object, nor the class symbol. And these rules must be applicable to objects of subclasses. This leads to the use of another form of rewrite rules which expresses changes only to some attributes.

Definition 39. Given an XGOTA signature $\Sigma = (S, \leqslant, A, F)$, we define an *auto-$\Sigma$-rewrite rule* with label $d \in F$, written as

$$d : f(t_1,\ldots,t_n) \to [ld] \Leftarrow C$$

where d is a declaration $f : s_1\{G_1\},\ldots,s_n\{G_n\} \to auto\{G\}$, ld is a Σ-description with minimum genetic pattern G, $t_i \in T_\Sigma(V)$ is a simple term with genome $s_i\ \{G_i\}$, for $i = 1,\ldots,n$, and t_1 is not a variable, $f(t_1,\ldots,t_n)$ is linear – there are not two occurrences of the same variable – $\mathbf{fvar}(ld) \subseteq \mathbf{fvar}(f(t_1,\ldots,t_n))$, and C is a finite set (possibly empty) of conditions of the form $e == e'$, where $e, e' \in E_\Sigma(V)$.

Definition 40. Given an XGOTA signature $\Sigma = (S, \leqslant, A, F)$, we define a *$\Sigma$-program R* as a set of Σ-rewrite rules, and auto-Σ-rewrite rules. We define an *XGOTA specification* as a pair $E = (\Sigma, R)$ where Σ is an XGOTA signature, and R is a Σ-program.

Example 41. For the situation in Example 6, we can define the following operations:

```
do: RussMult{times: Nat} → RussMult{times:Ok}.
rl do(L) → do(by2(L))
      ⇐ even(sayTimes(L))==true, (sayTimes(L) > 1)==true.
rl do(L) → do(plus(by2(L),sayTimes(L)))
      ⇐ even(sayTimes(L))==false, (sayTimes(L) > 1)==true.
rl do(L) → rest(L)
      ⇐ sayTimes(L)==1.
rl do(L) → rest(setNumber(L,0))
      ⇐ sayTimes(L)==0.
```

which reflect the behavior of Russian multipliers. In examples, keyword rl declares a rewrite rule labeled with the preceding declaration. They rely on the auxiliary

operations

```
by2:RussMult{times:Nat} → auto{ }.
rl  by2(RussMult[number ⇒ N1,  times ⇒ N2]) →
            [number ⇒ N1*2,  times ⇒ N2/2].
* * *
plus:RussMult{times:Nat}Nat{ } → auto{ }.
rl  plus(RussMult[number ⇒ N1],  N2) → [number ⇒ N1+N2].
* * *
sayTimes:RussMult{times:Nat} → Nat{ }.
rl  sayTimes(RussMult[times ⇒ N]) → N.
* * *
rest:RussMult{times:Nat} → auto{ }.
rl  rest(RussMult[times ⇒ N]) → [times ⇒ Ok[]].
```

We have assumed that some arithmetic and boolean operations, and constants like: +, *, /, >, even, true, and false are already specified. Since in XGOTA signatures values are objects, the value of class Ok is represented by the term Ok[]. Now, to operate with Russian multipliers we can set the number and times attributes with operations:

```
setNumber:Mult{ } → auto{ }.
rl  setNumber(Mult[ ],  N) → [number ⇒ N].
* * *
setTimes:RussMult{ } → auto{times:Nat}.
rl  setTimes(RussMult[ ],  T) → [times ⇒ T].
```

and after applying do, obtain the result accessing attribute number:

```
sayNumber:Mult{number:Nat} → Nat{ } .
rl  sayNumber(Mult[number ⇒ N]) → N.
```

We can put it in a procedure operation

```
multRuss:Nat{ }Nat{ } → Nat{ }.
rl  multRuss(N1,  N2) →
            sayNumber(do(setTimes(setNumber(RussMult[ ],  N1),  N2))).
```

where Russian multipliers are used both as data structure and control procedure.

6.1. Rewrite calculus

Now, we need a way to represent the effect of applying an operation. This corresponds to using the rewrite rules over expressions, which must be related with the definition of models we want for our XGOTA specifications. We will follow the idea in [20], considering an XGOTA specification as a theory in a special proof system,

where the application of rewrite rules are formulae that can be derived in such system from a given specification. Therefore, given an XGOTA specification $E=(\Sigma,R)$, our calculus should be able to derive *reduction statements* of the form $e \to e'$, expressing that the rules in R make possible to reduce expression e to expression e'. We will also need to derive strict equations of the form $e == e'$, expressing that the rules in R make possible to reduce expressions e, and e' to some common simple term t.

To deal with lazy evaluation, we will consider augmenting any XGOTA signature with a symbol representing an undefined value. This will be helpful for considering approximations of values, when the meaning of any non-relevant subexpression is dropped. Consequently, in any extended algebra that "undefined" symbol will have as meaning the bottom element of the CPO acting as carrier set. Abusing of the notation, we will use the same symbol $\bot$, conscious of the relation $[\![\bot]\!]^{\mathscr{A}}_{\alpha} = \bot$, and assuming that the context will allow to distinguish them. As an expression, $\bot$ must fulfill every genome, since having no information it might represent any possible value.

Due to the introduction of $\bot$, we will have the possibility to derive in our calculus reduction statements of the form $e \to t$, where t is a simple feature term possibly including occurrences of $\bot$. The intended meaning of such statement is "t is a finite approximation of e's value". As we will see in Section 7, the limit value of all the t such that $e \to t$ can be proved will characterize the semantic value of e, which may be an infinite object.

6.1.1. Static labeling for static typing

As we have seen while studying substitutions, there can be a difference between the minimum genome of a term, and the minimum genome of its instances. Since rewrite rules are applied by means of instances, this could lead to some problems regarding the minimal declaration of certain operations. Specifically, if the application of some rule makes the argument of a given operation to have a more specialized minimum genome, then it is possible that the minimal declaration becomes a more refined one. We can see it in the next example.

Example 42. Let us consider an XGOTA specification where we have the following declaration and rule for some operation f:

```
op  f:A{ } → C{ }.
rl  f(X) → C[g ⇒ X].
```

Observe that the right-hand side of the rule has more "genetic" information than the genome in the declaration codomain. This makes that an expression like f(A[]) has minimum genome C{ }, which is the one obtained by the declaration. Nevertheless, when reducing that expression with the corresponding rule we obtain C[g ⇒ A[]] which, of course, fulfills genome C{ }, but whose minimum genome is C{g:A}. This could be a complication when we use the original expression as argument for some operation with declarations which are sensitive to those changes in the genome. We consider an

operation h with declarations:

$$\mathsf{op}\ \ \mathsf{h{:}C}\{\,\} \to \mathsf{Nat}\{\,\}.$$
$$\ldots$$
$$\mathsf{op}\ \ \mathsf{h{:}C}\{\mathsf{g{:}A}\} \to \mathsf{Bool}\{\,\}.$$

Note that the second declaration for h is an instance of *non-covariant overriding*. Now, the expression h(f(A[])) has minimum genome Nat{ } because we use the minimal declaration for the minimum (static) genome of the argument. But, if we reduce the argument we get h(C[g ⇒ A[]]) which is an expression with minimum genome Bool{ } because now the minimal declaration of h to be used is the second one. This could pose a problem when deriving reduction statements with the calculus. Clearly, we could have

$$\mathsf{h(f(A[]))} \to \mathsf{h(C[g \Rightarrow A[]])}$$

by the rewrite rule given for f, and also

$$\mathsf{h(C[g \Rightarrow A[]])} \to \mathsf{Bool[\ldots]}$$

with some rule associated to the second declaration of h, hence having by transitivity of reduction that

$$\mathsf{h(f(A[]))} \to \mathsf{Bool[\ldots]}$$

This is a reduction which is not compatible with the meaning obtained with Definition 36, since we must use the first declaration given for h because that was the declaration used to construct the expression. We can obtain an adequate reduction if we label operations in expressions with the declaration we intend to use to obtain its meaning, and we stick to that declaration through reductions. If we name

d as $\mathsf{op\ \ h{:}C}\{\,\} \to \mathsf{Nat}\{\,\}$.
d' as $\mathsf{op\ \ h{:}C}\{\mathsf{g{:}A}\} \to \mathsf{Bool}\{\,\}$.

it would be

$$d{:}\mathsf{h(f(A[]))} \to d{:}\mathsf{h(C[g \Rightarrow A[]])}$$

having (because of the label d)

$$d{:}\mathsf{h(C[g \Rightarrow A[]])} \to \mathsf{Nat[\ldots]}$$

and by transitivity,

$$d{:}\mathsf{h(f(A[]))} \to \mathsf{Nat[\ldots]}$$

which is the proper reduction.

In this work we will concentrate on *static typing* for expressions. This means that the operation rules we want to apply are those corresponding to the minimum genome of the initial expression, no matter if there is a change in the minimum genome during the reduction. This precludes our formalism to model dynamic dispatch. Our decision

to allow a liberal overriding policy (restricted only by the regularity condition from Definition 23) and the deterministic denotation of expressions (Definition 36) makes impossible to dynamically select the lower operation declaration applicable to an expression without compromising the compile-time genome type of the expression as we have shown in Example 42.

To keep track of the static genome of an expression, there will be a slight increment of notation, including in operation expressions – just for derivations – the label corresponding to the declaration used to form them, and a complete description of attributes for feature expressions. This leads to the notion of *labeled expression*:

Definition 43. Given an XGOTA signature $\Sigma=(S,\leqslant,A,F)$, the set of *labeled Σ-expressions with genome $s\{G\}$* is inductively defined as follows:

- $\perp$.
- $x\in V$ if $\mathbf{mg}(x)\leqslant s\{G\}$.
- $w[g_1\Rightarrow e_1,\ldots,g_n\Rightarrow e_n]$ if $w\in S$, $\langle g_1,\ldots,g_n\rangle$ is the ordered set of attributes of w, there exist declarations for w, $g_i: w_i\rightarrow s_i\in A$, every e_i is a labeled Σ-expression with genome $v_i\{G_i\}\leqslant s_i\{\,\}$, for $i=1,\ldots,n$, and $w\{(g_1,v_1),\ldots,(g_n,v_n)\}\leqslant s\{G\}$.
- $x: w[g_1\Rightarrow e_1,\ldots,g_n\Rightarrow e_n]$ if $\mathbf{mg}(x)\leqslant s\{G\}$, and $w[g_1\Rightarrow e_1,\ldots,g_n\Rightarrow e_n]$ is a labeled Σ-expression with genome $\mathbf{mg}(x)$.
- $d: f(e_1,\ldots,e_n)$ if every e_i is a labeled Σ-expression with genome $w_i\{G_i\}$, for $i=1,\ldots,n$, and d is a declaration of f applicable to $w_1\{G_1\}\cdots w_n\{G_n\}$, such that:
  - if d is $f: w_1'\{G_1'\}\cdots w_n'\{G_n'\}\rightarrow w'\{G'\}$, then $w'\{G'\}\leqslant s\{G\}$, and
  - if d is $f: w_1'\{G_1'\}\cdots w_n'\{G_n'\}\rightarrow auto\{G'\}$, then $w_1\{G_1\leftarrow G'\}\leqslant s\{G\}$.

With labeled expressions we do not care about a minimum genome because troublesome expressions (those with operations) are annotated with the proper declaration. Besides, expression $\perp$ does not have a minimum genome. We will use $LE_\Sigma(X)$ to denote the set of labeled Σ-expressions constructed with free variables of X. We will also use $LT_\Sigma(X)$ to denote the set of labeled terms. We can obtain the corresponding labeled expression for any expression e, just by completing (and re-arranging) the attribute description of a feature expression, and annotating any operation subexpression with the corresponding minimal declaration. We will denote by $\mathbf{sl}(e)$, the *static labeling* of an expression e.

Substitutions applied to labeled expressions work as for expressions, but now, substitutions always make sense, obtaining well-formed labeled expressions. Substitutions which replace variables with simple labeled terms will be called *simple substitutions*, which are the ones we will use to apply rewrite rules.

Basically, annotated declarations are syntactic information, but they make a difference when dealing with the meaning of a labeled operation expression. In that case, we want to use the annotated declaration to denote the expression.

Definition 44. Given an XGOTA signature Σ, an XGOTA Σ-algebra $\mathscr{A}$, a labeled expression $d: f(e_1,\ldots,e_n)\in LE_\Sigma(X)$, and an $\mathscr{A}$-assignment for X α, the *denotation of*

$d : f(e_1, \ldots, e_n)$ *in* $\mathscr{A}$ *under* α is defined as follows:

$$[\![d : f(e_1, \ldots, e_n)]\!]_\alpha^{\mathscr{A}} =_{\text{def}} f_d^{\mathscr{A}}([\![e_1]\!]_\alpha^{\mathscr{A}}, \ldots, [\![e_n]\!]_\alpha^{\mathscr{A}})$$

With Definition 44, we can obtain the denotation of any labeled expression, provided that variables, and labeled feature expressions are denoted as indicated by Definition 36. Observe (i) that if $w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n] \in LE_\Sigma(X)$, then $[\![w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]]\!]_\alpha^{\mathscr{A}} = c_w^{\mathscr{A}}([\![e_1]\!]_\alpha^{\mathscr{A}}, \ldots, [\![e_n]\!]_\alpha^{\mathscr{A}})$, because attributes appear ordered and complete in the description part, and (ii) that a given expression e and its static labeling have the same meaning in a given algebra, and variables assignment.

Proposition 45. *Let* Σ *be an XGOTA signature,* $e \in LE_\Sigma(X)$ *a labeled expression, and* $\mathscr{A}$ *an XGOTA* Σ*-algebra. For every* $\mathscr{A}$*-assignment for* X, α, *it holds that if* e *fulfills genome* $s\{G\}$ *then* $[\![e]\!]_\alpha^{\mathscr{A}}$ *is well defined,* $[\![e]\!]_\alpha^{\mathscr{A}} \in (s\{G\})^{\mathscr{A}}$, *and the operator* $\Phi : (\mathbf{mg}(z))^{\mathscr{A}} \to (\mathbf{mg}(z))^{\mathscr{A}}$, *with* $\Phi(b) =_{\text{def}} [\![e]\!]_{\alpha[z/b]}^{\mathscr{A}}$, *is defined and continuous for* $z \in X$, *such that* $\mathbf{mg}(e) \leqslant \mathbf{mg}(z)$.

Proof. It is analogous to that of Proposition 37, considering the cases in which e is $\perp$, and $d : f(e_1, \ldots, e_n)$. □

Proposition 46. *Let* Σ *be an XGOTA signature,* $\mathscr{A}$ *an XGOTA* Σ*-algebra, and* $e \in E_\Sigma(X)$ *an expression, it holds that* $[\![e]\!]_\alpha^{\mathscr{A}} = [\![\mathbf{sl}(e)]\!]_\alpha^{\mathscr{A}}$, *for every* $\mathscr{A}$*-assignment for* X, α.

Proof. By structural induction over Σ-expression e, and using that Definition 36 always applies the denotation of the minimal declaration of operations, which coincides with the annotation in the static labeling of an expression. □

When using totally defined assignments, the denotation of simple labeled terms without $\perp$, is a totally defined value in the CPO carrier set of the algebra. This is shown in the following proposition.

Proposition 47. *Let* Σ *be an XGOTA signature,* $t \in LT_\Sigma(X)$ *a simple labeled term without* $\perp$, *and* $\mathscr{A}$ *an XGOTA* Σ*-algebra. For every totally defined* $\mathscr{A}$*-assignment for* X, α *it holds that* $[\![t]\!]_\alpha^{\mathscr{A}} \in \mathbf{tot}(\mathscr{C}_{\mathscr{A}})$.

Proof (*Sketch*). By Definitions 32 and 33, because simple labeled terms without $\perp$ are denoted by assignments, or semantic constructors $c_w^{\mathscr{A}}$ where every argument is filled in. □

We have a *restricted substitution lemma* for labeled expressions.

Lemma 48 (Restricted substitution lemma). *Let* Σ *be an XGOTA signature,* e *a labeled* Σ*-expression of* $LE_\Sigma(X)$, *and* $\mathscr{A}$ *an XGOTA* Σ*-algebra. For every* $\mathscr{A}$*-assignment*

for X, α, and every substitution $\theta : X \to LE_\Sigma(X)$, it holds that $[\![e\theta]\!]^{\mathcal{A}}_{\alpha} = [\![e]\!]^{\mathcal{A}}_{\alpha\theta}$, where $\alpha\theta$ is the variables assignment defined as $(\alpha\theta)(x) =_{\text{def}} [\![x\theta]\!]^{\mathcal{A}}_{\alpha}$, for any $x \in X$.

Proof (*Sketch*). It follows by structural induction over labeled expression e. □

Now, we will present the rules of the rewrite calculus. We start from an XGOTA specification $E = (\Sigma, R)$, and we will use letters e, e', and e_i to represent labeled expressions of $LE_\Sigma(V)$, while reserving t, t', and t_i for labeled terms of $LT_\Sigma(V)$. Descriptions will be denoted by ld. The presentation will proceed in three stages: rewrite rule instantiation, rewriting relation, and rewrite rule application.

6.1.2. Rewrite rule instantiation

The idea of the calculus in [20] is to derive reduction statements to represent rewriting reductions. The way to do that is to transform the left-hand side of a reduction statement until it matches an instance of a given rewrite rule. But in XGOTA specifications the application of rewrite rules is not only determined by the existence of an instance of a rewrite rule via a substitution.

Example 49. Let us consider the following XGOTA specification:

```
sorts A B.
subsort B < A.
att g1:A → Nat.
att g2:B → Nat.
var X:Nat.
op f:A{ } → Nat{ }.
rl f(A[g1 ⇒ X]) → 3*X.
```

(assuming sort Nat is previously declared with operations and constants). We see that operation f is declared, and defined with one rewrite rule. Term B[g1 ⇒ 6] is a particular case of A[g1 ⇒ 6]. Nevertheless, a matching mechanism based on substitutions will fail to match term B[g1 ⇒ 6] with the existing rule left-hand side because no substitution can replace the sort label. A similar problem arises when we extend the description of a given term with more attributes. No substitution can match term B[g2 ⇒ , 7 g1 ⇒ 6] with A[g1 ⇒ X] because no substitution can extend or reorder the attributes. In both cases, the operation f is applicable with the given declaration – which is minimal for the minimum genome of the term.

If we do not want to further complicate substitutions to deal with instances obtained by inheritance or extension, we could transform those terms with the calculus. The idea is to have rules describing how to obtain a *hierarchy instance*

$$\frac{}{w[ld] \to_{\text{h}} w'[ld]} \quad w' \leqslant w, \text{ and } w[ld], w'[ld] \in LT_\Sigma(V) \quad \text{(H1, Hierarchy 1)}$$

$$\frac{}{w[ld] \to_{\mathrm{h}} w[g \Rightarrow x \mid ld]} \quad w[g \Rightarrow x \mid ld], w[ld] \in LT_{\Sigma}(V),\ x \text{ fresh variable}$$
(H2, Hierarchy 2)

where the use of the notation $[g \Rightarrow x \mid ld]$ for the description makes the order of attributes in the description irrelevant.

$$\frac{}{w[ld] \to_{\mathrm{h}} w[ld']} \quad w[ld] \in T_{\Sigma}(V),\ ld' \text{ is the ordered permutation of } ld$$
(H3, Hierarchy 3)

The direction of the arrow in rules (H1)–(H3) indicates we can obtain a hierarchy instance in the right-hand side of $\to_{\mathrm{h}}$, from what we have in the left-hand side.

We are going to define the *set of instances* of a given set of rewrite rules R. To begin with we can consider the most simple instance of a rewrite rule, which is to obtain a rewrite rule from the specification. However, this step must be restricted to produce rewrite rules with labeled expressions.

$$\frac{}{d : f(t_1, \ldots, t_n) \to \mathbf{sl}(r) \Leftarrow \mathbf{sl}(C)} \quad d : f(t_1, \ldots, t_n) \to r \Leftarrow C \in R \quad \text{(A, Axiom)}$$

For a condition C, **sl** is applied to every equation $e == e'$ in C as $\mathbf{sl}(e) == \mathbf{sl}(e')$, obtaining a condition $\mathbf{sl}(C)$. This rule works for auto-rewrite rules as well, considering r as the corresponding description, and applying operator **sl** to every attribute in a description. Now, we can obtain an instance of a rewrite rule via a substitution.

$$\frac{d : f(t_1, \ldots, t_n) \to r \Leftarrow C}{d : f(t_1, \ldots, t_n)\theta \to r\theta \Leftarrow C\theta} \quad \theta \text{ simple } \Sigma\text{-substitution} \quad \text{(S, Substitution)}$$

The next transformation has to deal with the hierarchy; we make the rewrite rules less general to adapt them to particular cases.

$$\frac{t'_1 \to_{\mathrm{h}} t_1 \cdots t'_n \to_{\mathrm{h}} t_n \quad d : f(t'_1, \ldots, t'_n) \to r \Leftarrow C}{d : f(t_1, \ldots, t_n) \to r \Leftarrow C} \quad \text{(P, Propagation)}$$

Observe that we restrict the transformation of the arguments of the left-hand side of the rewrite rule to those obtained with $\to_{\mathrm{h}}$. This guarantees that only instances corresponding to rules via (H1)–(H3) are produced.

Definition 50. Given an XGOTA specification $E = (\Sigma, R)$, a *Σ-hierarchy form* of R is any rewrite rule obtained from R applying rules (H1)–(H3), (A), and (P). We define the set of *Σ-instances* of R, noted by $[R]_{\mathrm{h}}$, as the set of rewrite rules obtained applying rule (S) to a hierarchy form of R.

In rule (S) we require substitution θ to be simple in order to guarantee that instances have left-hand side patterns formed by simple terms.

6.1.3. Rewriting relation

The following rules describe the behavior of the rewriting relation:

$$\frac{}{e \to e} \quad \text{(X, Reflexivity)}$$

$$\frac{e \to e' \;\; e' \to e''}{e \to e''} \quad \text{(T, Transitivity)}$$

$$\frac{e_1 \to e_1' \cdots e_n \to e_n'}{w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n] \to w[g_1 \Rightarrow e_1', \ldots, g_n \Rightarrow e_n']}$$
$$w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n] \in LE_\Sigma(V) \quad \text{(MN1, Monotonicity 1)}$$

$$\frac{e_1 \to e_1' \cdots e_n \to e_n'}{d : f(e_1, \ldots, e_n) \to d : f(e_1', \ldots, e_n')} \quad d : f(e_1, \ldots, e_n) \in LE_\Sigma(V)$$
(MN2, Monotonicity 2)

The *bottom* rule expresses the fact that $\perp$ is a (trivial) finite approximation of the value of any expression. In combination with the other rules in the calculus, this helps to reflect the behavior of lazy evaluation.

$$\frac{}{e \to \perp} \quad \text{(B, Bottom)}$$

The *join* rule states that for two expressions e, e' to be strictly equal, a Σ-term t (with totally defined denotation) must exist such that both can be reduced to it:

$$\frac{e \to t \;\; e' \to t}{e == e'} \quad t \in LE_\Sigma(V) \text{ simple term without } \perp \quad \text{(J, Join)}$$

Finally, we need a rule that captures the recursive meaning of tagged expressions:

$$\frac{}{e \to w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]\{x \mapsto e\}}$$
$$e \text{ is } x : w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n] \in LE_\Sigma(V) \quad \text{(U, Unfolding)}$$

Observe that if x does not occur in $w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]$ then the tag (i.e. the recursion) disappears after applying rule (U).

6.1.4. Rewrite rule application

First rule for reduction covers the standard term rewriting mechanism:

$$\frac{C}{d : f(t_1, \ldots, t_n) \to e} \quad d : f(t_1, \ldots, t_n) \to e \Leftarrow C \in [R]_h$$
(SR, Standard Reduction)

Observe that we only allow to reduce an operation term with an instance of a rule of the same declaration used to construct it. Auto-reduction is a more refined mechanism, we have to *transform* the first argument of the operation accordingly to the partial description that appears at the right-hand side of the method rewrite rule.

Definition 51. Given an XGOTA signature Σ, and Σ-descriptions ld and $g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m$ for s, we define the *transformation* denoted as $ld \leftarrow (g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m)$, as

the ordered description ld' such that $g \Rightarrow e$ is in ld' if and only if one of the following conditions holds:

- $g \equiv g_j$, for some $j \in \{1,\ldots,m\}$, and $e \equiv e_j$, or
- $g \not\equiv g_j$, for every $j = 1,\ldots,m$, and $g \Rightarrow e$ is in ld.

Definition 52. Given an XGOTA signature Σ, a labeled feature Σ-term $w[ld]$ with sort s, and a Σ-description ld' for s, the *transformation* of $w[ld]$ by ld', denoted as $(w[ld])[ld']$, is defined as $w[ld \leftarrow ld']$ if and only if it is a labeled Σ-expression.

The calculus rule for auto-rewriting is as follows:

$$\frac{C}{\begin{array}{l} d : f(t_1,\ldots,t_n) \to (t_1)[g_1 \Rightarrow e_1,\ldots,g_m \Rightarrow e_m] \\ \quad d : f(t_1,\ldots,t_n) \to [g_1 \Rightarrow e_1,\ldots,g_m \Rightarrow e_m] \Leftarrow C \in [R]_{\mathrm{h}} \\ \quad \text{and } (t_1)[g_1 \Rightarrow e_1,\ldots,g_m \Rightarrow e_m] \in LE_\Sigma(X) \quad (\mathrm{AR},\ \text{Auto-Reduction}) \end{array}}$$

The rules we have just presented are gathered in the *static XGOTA rewrite calculus.*

Definition 53. Given an XGOTA specification $E = (\Sigma, R)$, we define the *static XGOTA rewrite calculus*, or SXRC for short, as the calculus composed by rules (X), (T), (MN1), (MN2), (B), (J), (U), (SR), and (AR).

Definition 54. Given an XGOTA specification $E = (\Sigma, R)$, we say that a reduction statement, $e \to e'$, or a strict equation $e == e'$, *is derived* (*or proven*) *in SXRC* from the rules of R, denoted as

$$R \vdash_{\mathrm{SXRC}} e \to e' \quad \text{respectively } R \vdash_{\mathrm{SXRC}} e == e'$$

if the reduction statement, respectively the strict equation, can be inductively constructed with the rules of SXRC from the set of instances of R, $[R]_{\mathrm{h}}$.

Example 55. Let us consider the following XGOTA specification portraying lists:

```
sorts List NEList Nil.
subsort NEList Nil < List.
att head:NEList → Nat.
att tail:NEList → List.
var X:Nat.
op head:NEList{ } → Nat{ }.  *** declaration d
rl head(NEList[head ⇒ X]) → X.
op forever:Nat{ } → NEList{ }.  *** declaration d'
rl forever(X) → NEList[head ⇒ X, tail ⇒ forever(X)].
```

(assuming sort Nat is previously declared with operations and constants). Operation head consults the first element of a non-empty list, while forever constructs an infinite

list with the element given as argument. Reducing expression head(forever(5)), begins by considering the static labeling

d:head(d':forever(5))

and then obtaining an instance for d':forever(5) using rules (A), and (S) with substitution $\theta = \{\mathsf{X} \mapsto 5\}$.

d':forever(5)→NEList[head ⇒ 5, tail ⇒ d':forever(5)]

By rules (SR) and (MN2) we get

$$R \vdash_{\mathrm{SXRC}} d\colon \mathsf{head}(d'\colon \mathsf{forever}(5)) \to d\colon \mathsf{head}(\mathsf{NEList}[\mathsf{head} \Rightarrow 5,\ \mathsf{tail} \Rightarrow d'\colon \mathsf{forever}(5)])$$

Now, we cannot apply directly the rule for head, since the actual argument has tail while the formal does not. We have to apply rule (H2) to obtain a hierarchy instance

NEList[head ⇒ 5]→$_{\mathrm{h}}$NEList[head ⇒ 5, tail ⇒ Y]

and by (P) and (S) with $\theta' = \{\mathsf{X} \mapsto 5, \mathsf{Y} \mapsto \bot\}$, the instance

d:head(NEList[head ⇒ 5, tail ⇒ ⊥])→5

The argument is treated with rules (B), (MN1) and (MN2) for

$$R \vdash_{\mathrm{SXRC}}\ d'\colon \mathsf{forever}(5) \to \bot$$
$$R \vdash_{\mathrm{SXRC}} \mathsf{NEList}[\mathsf{head} \Rightarrow 5, \mathsf{tail} \Rightarrow d'\colon \mathsf{forever}(5)] \to \mathsf{NEList}[\mathsf{head} \Rightarrow 5, \mathsf{tail} \Rightarrow \bot]$$
$$R \vdash_{\mathrm{SXRC}} d\colon \mathsf{head}(\mathsf{NEList}[\mathsf{head} \Rightarrow 5,\ \mathsf{tail} \Rightarrow d'\colon \mathsf{forever}(5)]) \to d\colon \mathsf{head}(\mathsf{NEList}[\mathsf{head} \Rightarrow 5,\ \mathsf{tail} \Rightarrow \bot])$$

and then applying the instance in (SR) and using (T) we obtain

$$R \vdash_{\mathrm{SXRC}} d\colon \mathsf{head}(d'\colon \mathsf{forever}(5)) \to 5$$

7. Models and canonical tree model

Given an XGOTA specification $E = (\Sigma, R)$, a *model* is any XGOTA algebra for Σ, that behaves as the rules in R determine. We define *satisfaction*, noted with $\models$.

Definition 56. Given an XGOTA specification $E = (\Sigma, R)$, and an XGOTA Σ-algebra $\mathscr{A}$, we say that

- $(\mathscr{A}, \alpha) \models e \to e'$, where $e, e' \in LE_\Sigma(X)$, an α is an $\mathscr{A}$-assignment for X, if and only if $[\![e']\!]^{\mathscr{A}}_{\alpha} \sqsubseteq [\![e]\!]^{\mathscr{A}}_{\alpha}$.

- $(\mathscr{A},\alpha)\models e==e'$, where $e,e'\in LE_\Sigma(X)$, and α is an $\mathscr{A}$-assignment for X, if and only if there exists a totally defined element $a\in\mathbf{tot}(C_\mathscr{A})$, such that $[\![e]\!]^\mathscr{A}_\alpha = a = [\![e']\!]^\mathscr{A}_\alpha$.
- $\mathscr{A}\models d: f(t_1,\ldots,t_n)\to e\Leftarrow C$, a hierarchy form of a Σ-rewrite rule if and only if for every $\mathscr{A}$-assignment for the variables in $\mathbf{fvar}(f(t_1,\ldots,t_n))$, α, such that $(\mathscr{A},\alpha)\models C$, it holds

$$(\mathscr{A},\alpha)\models d:\ f(t_1,\ldots,t_n)\to e$$

- $\mathscr{A}\models d: f(t_1,\ldots,t_n)\to[g_1\Rightarrow e_1,\ldots,g_m\Rightarrow e_m]\Leftarrow C$, a hierarchy form of an auto-Σ-rewrite rule if and only if $(t_1)[g_1\Rightarrow e_1,\ldots,g_m\Rightarrow e_m]$ is an expression, and for every $\mathscr{A}$-assignment for the variables in $\mathbf{fvar}(f(t_1,\ldots,t_n))$, α, such that $(\mathscr{A},\alpha)\models C$, it holds

$$(\mathscr{A},\alpha)\models d:\ f(t_1,\ldots,t_n)\to(t_1)[g_1\Rightarrow e_1,\ldots,g_m\Rightarrow e_m]$$

- $\mathscr{A}$ is a *model* of E, noted as $\mathscr{A}\models E$, if and only if $\mathscr{A}$ satisfies every hierarchy form of R.

Note that $(\mathscr{A},\alpha)\models C$, corresponds to $\mathscr{A}$ satisfying every strict equation in C under α. Not every XGOTA specification may have models since this depends on the satisfiability of every hierarchy form. If there exist rules with the same left-hand side but inconsistent right-hand ones (just by having different sort constructor) no algebra could be model. Distinctly from [20] where non-deterministic functions are considered, we will restrict ourselves to a special kind of programs for which models (and initial model) exist, as it is shown later on. Given an XGOTA specification $E=(\Sigma,R)$, the calculus SXRC is sound in the sense of having that every reduction or strict equation derived in SXRC from R is satisfied in every model of E.

Theorem 57 (SXRC soundness). *Given an XGOTA specification $E=(\Sigma,R)$, and an XGOTA Σ-algebra $\mathscr{A}$, model of E, for every reduction statement or strict equation ϕ, if $R\vdash_{\mathrm{SXRC}}\phi$ then for every totally defined $\mathscr{A}$-assignment α, $(\mathscr{A},\alpha)\models\phi$.*

Proof. Let α be any totally defined $\mathscr{A}$-assignment. We have to show that if ϕ can be derived from the rewrite rules of R using the calculus SXRC then $\mathscr{A}$ satisfies ϕ, that is $(\mathscr{A},\alpha)\models\phi$. Since the calculus SXRC works inductively to derive reduction statements or strict equations, we can proceed by structural induction over SXRC derivations. We distinguish cases according to the last SXRC rule used in a given derivation for ϕ:

- Rules (X), (T), and (B) hold trivially by properties of $\sqsubseteq$. Rules (MN1) and (MN2) hold by the monotonicity of the denotations $c^\mathscr{A}_w$, and $f^\mathscr{A}_d$, while rule (J) holds by Proposition 47 and induction hypothesis applied to the rule premises.
- *Unfolding rule*: We have that ϕ is $e\to w[g_1\Rightarrow e_1,\ldots,g_n\Rightarrow e_n]\{x\mapsto e\}$, where e is $x: w[g_1\Rightarrow e_1,\ldots,g_n\Rightarrow e_n]$. By Definition 36, $[\![e]\!]^\mathscr{A}_\alpha=\mathbf{fix}(\Phi)$, where $\Phi:(\mathbf{mg}(x))^\mathscr{A}\to(\mathbf{mg}(x))^\mathscr{A}$ is the operator

$$\Phi(b)=[\![w[g_1\Rightarrow e_1,\ldots,g_n\Rightarrow e_n]]\!]^\mathscr{A}_{\alpha[x/b]}.$$

Now, since e is labeled, $w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]\{x \mapsto e\}$ also is, and it holds

$$\begin{aligned}
&[\![w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]\{x \mapsto e\}]\!]^{\mathscr{A}}_{\alpha} \\
&\quad = [\![w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]]\!]^{\mathscr{A}}_{\alpha\{x \mapsto e\}} \quad (\textit{Lemma } 48) \\
&\quad = [\![w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]]\!]^{\mathscr{A}}_{\alpha[x/[\![e]\!]^{\mathscr{A}}_{\alpha}]} \quad (\textit{Definition } 34, \textit{ and} \\
&\qquad\qquad [\![x\{x \mapsto e\}]\!]^{\mathscr{A}}_{\alpha} = [\![e]\!]^{\mathscr{A}}_{\alpha}) \\
&\quad = [\![w[g_1 \Rightarrow e_1, \ldots, g_n \Rightarrow e_n]]\!]^{\mathscr{A}}_{\alpha[x/\mathbf{fix}(\Phi)]} \\
&\quad = \Phi(\mathbf{fix}(\Phi)) \\
&\quad = \mathbf{fix}(\Phi) \\
&\quad = [\![e]\!]^{\mathscr{A}}_{\alpha}
\end{aligned}$$

hence $(\mathscr{A}, \alpha) \models \phi$.

- *Standard reduction rule*: We have that ϕ has been obtained from an instance of a standard rule of R. That is, there exists a hierarchy form $d : f(t_1, \ldots, t_n) \to e \Leftarrow C$, and a simple substitution θ, such that ϕ is $d : f(t_1, \ldots, t_n)\theta \to e\theta$ obtained from $C\theta$, a set of strict equations. By structural induction hypothesis it holds $(\mathscr{A}, \alpha) \models C\theta$, and having that conditions in hierarchy forms are composed by labeled expressions, we can apply Lemma 48, obtaining $(\mathscr{A}, \alpha\theta) \models C$. With that result, since $\mathscr{A}$ is a model, we have

$$(\mathscr{A}, \alpha\theta) \models d : f(t_1, \ldots, t_n) \to e$$

which means that $[\![e]\!]^{\mathscr{A}}_{\alpha\theta} \sqsubseteq [\![d : f(t_1, \ldots, t_n)]\!]^{\mathscr{A}}_{\alpha\theta}$. Again, as the right-hand sides of hierarchy forms are labeled expressions we can apply Lemma 48, having $[\![e\theta]\!]^{\mathscr{A}}_{\alpha} \sqsubseteq [\![d : f(t_1, \ldots, t_n)\theta]\!]^{\mathscr{A}}_{\alpha}$, hence $(\mathscr{A}, \alpha) \models \phi$.

- *Auto-reduction rule*: We have that ϕ has been obtained from an instance of an auto-rule of R. That is, there exists an auto-hierarchy form $d : f(t_1, \ldots, t_n) \to [g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m] \Leftarrow C$, and a simple substitution θ, such that ϕ is $d : f(t_1, \ldots, t_n)\theta \to (t_1\theta)[g_1 \Rightarrow e_1\theta, \ldots, g_m \Rightarrow e_m\theta]$ obtained from $C\theta$, a set of strict equations. By structural induction hypothesis it holds $(\mathscr{A}, \alpha) \models C\theta$, and having that conditions in hierarchy forms are composed by labeled expressions, we can apply Lemma 48, obtaining $(\mathscr{A}, \alpha\theta) \models C$. Since $\mathscr{A}$ is a model, if $(t_1)[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]$ is a term, then we have

$$(\mathscr{A}, \alpha\theta) \models d : f(t_1, \ldots, t_n) \to (t_1)[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]$$

which means that $[\![(t_1)[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]]\!]^{\mathscr{A}}_{\alpha\theta} \sqsubseteq [\![d : f(t_1, \ldots, t_n)]\!]^{\mathscr{A}}_{\alpha\theta}$. Now, given that t_1 is a feature term by Definition 39, then

$$(t_1)[g_1 \Rightarrow e_1, \ldots, g_m \Rightarrow e_m]\theta \equiv (t_1\theta)[g_1 \Rightarrow e_1\theta, \ldots, g_m \to e_m\theta]$$

and having that the right-hand sides of auto-hierarchy forms are labeled expressions, by Lemma 48, it holds $[\![(t_1\theta)[g_1 \Rightarrow e_1\theta, \ldots, g_m \Rightarrow e_m\theta]]\!]^{\mathscr{A}}_{\alpha} \sqsubseteq [\![d : f(t_1, \ldots, t_n)\theta]\!]^{\mathscr{A}}_{\alpha}$, hence $(\mathscr{A}, \alpha) \models \phi$. $\square$

Observe that the hypothesis of assignment α being totally defined is essential because otherwise we could not show satisfiability of strict equations which are provable. Consider $x = = x$ which is derived in SXRC from any program R by applying rules (R) and (J) since $x \in LT_\Sigma(V)$ simple without $\perp$. For any model $\mathscr{A}$ of a specification with R, $[\![x]\!]^{\mathscr{A}}_\alpha = \alpha(x)$ which may not be totally defined if α is not totally defined, thus having that $(\mathscr{A}, \alpha) \not\models x = = x$.

7.1. Canonical tree model

Given a GOTA signature $\Sigma = (S, \leqslant, A)$, we define an *extended feature tree* a as a mapping from a tree domain D_a (a prefix-closed subset of $(\mathbf{symb}(A))^*$), into the set $S \cup V$. Elements in a tree domain are referred as *tree positions*. Note that a non-empty tree domain always contains the empty chain, ε, which is known as the *root position*. Observe also that there exists a tree with an empty domain. We will denote that tree as $\perp$ (abusing again of the notation). We will use the notion of *subtree* at a given position, or node.

Definition 58. Given a GOTA signature $\Sigma = (S, \leqslant, A)$, an extended feature tree a, and a position u, the *subtree* of a *at position* u, noted by $a|_u$, is defined as the tree such that

$$D_{a|_u} =_{\text{def}} \{v \in \mathbf{symb}(A)^* \mid uv \in D_a\}, \quad a|_u(v) =_{\text{def}} a(uv).$$

Observe that if $u \notin D_a$, then $a|_u$ is $\perp$, which makes sense if we consider that the nodes in a reached through position u are not defined. We need feature trees which respect the signature. We can define the set of *canonical trees* of a given genome.

Definition 59. Given a signature $\Sigma = (S, \leqslant, A)$, the set of *canonical Σ-trees with genome s $\{G\}$*, is defined as the set of extended feature trees a such that one of the following conditions holds:

- $a = \perp$.
- If $a(\varepsilon) = x \in V$, then $\mathbf{mg}(x) \leqslant s\{G\}$, and $D_a = \{\varepsilon\}$.
- If $a(\varepsilon) = w \in S$, then $w \leqslant s$, and for any attribute symbol $g \in D_a$, there exists a declaration of g for w, $g : w' \to s' \in A$ such that $a|_g$ is a canonical tree of sort s', and for every pair $(g, v) \in G$, $a|_g$ is a canonical tree of sort v.

The set of canonical Σ-trees formed with variables of a set X is denoted by $CT_\Sigma(X)$. Canonical trees so defined can be assimilated with the abstract values of an XGOTA algebra. The sort symbol in the root represents the sort the element belongs to, and the children represent the values of the corresponding attributes, which are themselves canonical trees. The undefined positions can be thought as having value $\perp$. Since tree domains are possibly infinite we can represent any degree of recursion of sorts and attributes. Canonical trees respect the sort inclusion, depending on their tree positions

and the information they have in them. An *approximation ordering* can also be defined over $CT_\Sigma(X)$.

Definition 60. Given a signature $\Sigma=(S,\leqslant,A)$, and any set of variables $X\subseteq V$, the relation $\sqsubseteq$ over $CT_\Sigma(X)$ is defined as

$$a\sqsubseteq a' \Leftrightarrow_{\text{def}} \text{ for every } u\in D_a, \text{ it holds } u\in D_{a'}, \text{ and } a'(u)=a(u)$$

for any $a,a'\in CT_\Sigma(X)$.

Thus, given two canonical trees a, a', we say that $a\sqsubseteq a'$, if a' preserves the information in all the positions occurring in a, possibly adding more information in new positions. Relation $\sqsubseteq$ over $CT_\Sigma(X)$ is a partial order, which makes possible to define a CPO of canonical trees (as in [19]).

Proposition 61. *Given a signature* $\Sigma=(S,\leqslant,A)$, *and any set of variables* $X\subseteq V$, $(CT_\Sigma(X),\sqsubseteq,\perp)$ *is a CPO.*

Proof (*Sketch*). Given a directed set $Q\subseteq CT_\Sigma(X)$, there exists the least upper bound of Q, defined as the join of the canonical trees in Q when they are seen as relations:

$$\bigsqcup Q =_{\text{def}} \bigcup_{a\in Q} a.$$

It can easily be shown that the join is a canonical tree, an upper bound of Q, and a lower bound of any other upper bound. Therefore, the join is the least upper bound of Q, and $(CT_\Sigma(X),\sqsubseteq,\perp)$ is a CPO. □

We will also consider the set of finite domain canonical trees which are less than a given tree a in the approximation ordering. We will call it the *base* of a, $\mathbf{B}(a)$. It holds that finite canonical trees are the finite elements in the $CT_\Sigma(X)$ CPO, which along with the following properties:

- $\mathbf{B}(a)$ is directed, and $\bigsqcup \mathbf{B}(a)=a$, and
- $\mathbf{B}(\bigsqcup_{i\in I} a_i)=\bigcup_{i\in I}\mathbf{B}(a_i)$,

allows us to show that $(CT_\Sigma(X),\sqsubseteq,\perp)$ is an algebraic CPO.

Corollary 62. *Given a signature* $\Sigma=(S,\leqslant,A)$, *and any set of variables* $X\subseteq V$, $(CT_\Sigma(X),\sqsubseteq,\perp)$ *is an algebraic CPO.*

Proof. By Proposition 61, $(CT_\Sigma(X),\sqsubseteq,\perp)$ is a CPO, the set of finite elements is the set of finite trees which is countable, and every $a\in CT_\Sigma(X)$ is the least upper bound of its base, $\bigsqcup \mathbf{B}(a)=a$, which is a directed set of finite elements, having that $(CT_\Sigma(X),\sqsubseteq,\perp)$ is algebraic. □

Some other concepts that will be interesting are presented now. We consider the *pruning* of an extended feature tree at a given level.

Definition 63. Given a signature $\Sigma=(S,\leqslant,A)$, an extended feature tree a, and $k\in\mathbb{N}$, the *pruning of a at level k*, noted by $a\|_k$, is defined as the tree such that

$$D_{a\|_k} =_{\text{def}} \{v\in D_a \mid v \text{ has length less than } k\}, \quad a\|_k(v) =_{\text{def}} a(v).$$

To prune a tree at level k corresponds to discard any node with a depth greater than or equal to k, where the depth is the length of the position of a given node:

Proposition 64. *Given a signature* $\Sigma=(S,\leqslant,A)$, *and a canonical tree a, the following properties hold*:

(i) $a\|_0\equiv\bot$.
(ii) *If* $i\leqslant j$ *then* $a\|_i\sqsubseteq a\|_j$.
(iii) *If a is a tree of level* $k\in\mathbb{N}$, *then* $a\equiv a\|_l$ *for any* $l>k$.
(iv) *For any* $k\in\mathbb{N}$, $\bot\|_k\equiv\bot$.
(v) $\bigsqcup_{k\in\mathbb{N}} a\|_k = a$.

To ease the notation while defining the algebra of canonical trees, we introduce the auxiliary definitions of *extension* and *restriction*.

Definition 65. Given a signature $\Sigma=(S,\leqslant,A)$, and a relation $a\subseteq\mathbf{symb}(A)^*\times(S\cup V)$, we define

- the *extension* of a in $g\in\mathbf{symb}(A)$, and $s\in S$, $a\,|^g(s) =_{\text{def}} \{(gu,v)\,|\,(u,v)\in a\}\cup\{(\varepsilon,s)\}$.
- the *restriction* (subtree) of a in $g\in\mathbf{symb}(A)$, $a|_g =_{\text{def}} \{(u,v)\,|\,(gu,v)\in a\}$.

There is a one-to-one correspondence between canonical finite trees and simple feature terms. We can define the simple labeled Σ-term associated to a finite canonical tree a, $\mathbf{t}(a)$, and the canonical Σ-tree associated to a simple $t\in LT_\Sigma(X)$, noted as $\mathbf{a}(t)$, and it holds $\mathbf{a}(\mathbf{t}(a))=a$, and $\mathbf{t}(\mathbf{a}(t))=t$. Observe that the extension $a\,|^g(s)$ is the tree $\mathbf{a}(s[g\Rightarrow\mathbf{a}(a)])$

7.1.1. The algebra of canonical trees

Now we can define the XGOTA algebra of canonical trees, obtained considering the canonical trees of a given signature. But to define the denotation of operations, we have to use a set of rewrite rules for them, hence the algebra of canonical trees will be related to a program.

Definition 66. Given an XGOTA signature $\Sigma=(S,\leqslant,A,F)$, a Σ-program R, a set of variables $X\subseteq V$, and the set of canonical trees $CT_\Sigma(X)$, we define the XGOTA *Σ-algebra of canonical trees by R*, $\mathscr{CT}_{\Sigma,X}(R)$ as follows:

- The carrier set is $\mathscr{C}_{\mathscr{CT}_{\Sigma,X}(R)} =_{\text{def}} CT_\Sigma(X)$, the CPO of canonical trees.
- For every $s\,\{G\}$,

$$(s\{G\})^{\mathscr{CT}_{\Sigma,X}(R)} =_{\text{def}} \{a\in CT_\Sigma(X)\,|\,a \text{ is a tree with genome } s\{G\}\}$$

- For every sort s with n attributes ($n>0$), where $g_1,\ldots,g_n$ is the ordered set of attributes of s, for any $a_1,\ldots,a_n \in \mathscr{C}_{\mathscr{CT}_{\Sigma,X}(R)}$,

$$c_s^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n) =_{\text{def}} \begin{cases} \bigcup_{i=1,\ldots,n} a_i|^{g_i}(s) & \text{if there exists an ordered complete set of attributes of } s,\ g_i : s_i \to w_i \in A \text{ with } a_i \in w_i^{\mathscr{CT}_{\Sigma,X}(R)},\ \text{for } i=1,\ldots,n, \\ \bot & \text{otherwise.} \end{cases}$$

- For every sort s with no attributes, $c_s^{\mathscr{CT}_{\Sigma,X}(R)} =_{\text{def}} a$, where a is such that $D_a=\{\varepsilon\}$ and $a(\varepsilon)=s$.
- For every declaration d with domain $s_1\{G_1\}\cdots s_n\{G_n\}$ of an operation symbol f,

$$f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n) =_{\text{def}} \bigcup_{a\in B} a$$

for any $a_1 \in (s_1\{G_1\})^{\mathscr{CT}_{\Sigma,X}(R)},\ldots,a_n \in (s_n\{G_n\})^{\mathscr{CT}_{\Sigma,X}(R)}$, where

$$B = \{\mathbf{a}(t) \mid R \vdash_{\text{SXRC}} d : f(\mathbf{t}(b_1),\ldots,\mathbf{t}(b_n)) \to t, t \in LT_\Sigma(X) \text{ simple}\}$$

having $b_i \in \mathbf{B}(a_i),\ i=1,\ldots,n$.

Observe that to obtain a canonical tree, the set B has to admit a least upper bound, which is the meaning of calculating the join of the canonical trees in B. This could be not possible if B contains two incomparable trees, which is the case when two rewrite rules with "incomparable" right-hand sides can be used to obtain a reduction statement $d : f(\mathbf{t}(b_1),\ldots,\mathbf{t}(b_n)) \to t$. We will consider only *deterministic* programs.

Definition 67. Given an XGOTA signature $\Sigma=(S,\leqslant,A,F)$, and a Σ-program R, we will say that R is *deterministic* if for every declaration d of an operation f of Σ, with domain $s_1\{G_1\}\cdots s_n\{G_n\}$, the set

$$\{\mathbf{a}(t) \mid R \vdash_{\text{SXRC}} d : f(\mathbf{t}(b_1),\ldots,\mathbf{t}(b_n)) \to t, t \in LT_\Sigma(X) \text{ simple}\}$$

is directed, for any $a_1 \in (s_1\{G_1\})^{\mathscr{CT}_{\Sigma,X}(R)},\ldots,a_n \in (s_n\{G_n\})^{\mathscr{CT}_{\Sigma,X}(R)}$, and $b_i \in \mathbf{B}(a_i)$, $i=1,\ldots,n$.

This semantic condition is assumed here for technical convenience. However, decidable sufficient conditions for determinism can be formulated. For instance, one could demand that any two different rewrite rules whose left-hand sides can overlap are distinguished by different, incompatible conditions. This idea is made precise in [21], in the context of a simple functional language, and it can be easily adapted to XGOTA.

For any given deterministic program R, $\mathscr{CT}_{\Sigma,X}(R)$ just defined is an XGOTA algebra for signature Σ.

Theorem 68 ($\mathscr{CT}_{\Sigma,X}(R)$ is algebra). *Given an XGOTA signature $\Sigma=(S,\leqslant,A,F)$, a deterministic Σ-program R, and a set of variables $X\subseteq V$, the Σ-algebra of canonical trees by R, $\mathscr{CT}_{\Sigma,X}(R)$ is an XGOTA algebra for Σ.*

7.1.2. The algebra of canonical trees is a model

To show that the algebra of canonical trees is a model of the program by which it is defined we need some more technical results. We need to consider the *identity* $\mathscr{CT}_{\Sigma,X}(R)$-assignment, defined as $id(x) =_{\text{def}} \mathbf{a}(x)$, for any $x \in X$. Observe that, for every variable x, $id(x)$ is total, since it is $\mathbf{a}(x)$ which is a maximal tree in $CT_\Sigma(X)$. Note also that $id(x)$ is finite since $\mathbf{a}(x)$ has finite range for any $x \in X$, therefore, id is totally defined. We can state some important auxiliary results.

Proposition 69. *Given an XGOTA signature Σ, a deterministic Σ-program R, and any set of variables $X \subseteq V$, it holds $[\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = \mathbf{a}(t)$, for any simple term $t \in LT_\Sigma(X)$.*

Proposition 70. *Given an XGOTA signature Σ, a deterministic Σ-program R, and any set of variables $X \subseteq V$, it holds $R \vdash_{\text{SXRC}} t \to t' \Leftrightarrow [\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsupseteq [\![t']\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$, for any $t,\ t' \in LT_\Sigma(X)$ simple.*

Proof (*Sketch*). The proof relies on the fact that t, and t' are denoted by their associated tree by Proposition 69, proceeding by structural induction over t. □

Proposition 71. *Given an XGOTA signature Σ, a deterministic Σ-program R, and any set of variables $X \subseteq V$, and $e \in LE_\Sigma(X)$, it holds*

$$[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = \bigsqcup\{\mathbf{a}(t) \mid R \vdash_{\text{SXRC}} e \to t,\ t \in LT_\Sigma(X) \text{ simple}\}.$$

Proof (*Sketch*). If we denote by $C(e)$ the set $\{\mathbf{a}(t) \mid R \vdash_{\text{SXRC}} e \to t,\ t \in LT_\Sigma(X) \text{ simple}\}$, then the proof can proceed by structural induction over expression e, analyzing set $C(e)$. In every case the form of e and the existing rules in SXRC determine $C(e)$ to be such that $[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = \bigsqcup C(e)$. A complete demonstration can be found in [32]. □

In the previous proof we worked on expressions determining possible reductions given the existing rules in the calculus SXRC. Next we show that satisfaction in $\mathscr{CT}_{\Sigma,X}(R)$ can be characterized in terms of SXRC derivability.

Lemma 72 (Characterization Lemma). *Given an XGOTA signature Σ, a deterministic Σ-program R, and any set of variables $X \subseteq V$,*

(i) *For any reduction statement $e \to t$, with $e \in LE_\Sigma(X),\ t \in LT_\Sigma(X)$ simple,*

$$(\mathscr{CT}_{\Sigma,X}(R), id) \models e \to t \Leftrightarrow R \vdash_{\text{SXRC}} e \to t$$

(ii) *For any strict equation $e == e'$, with $e, e' \in LE_\Sigma(X)$,*

$$(\mathscr{CT}_{\Sigma,X}(R), id) \models e == e' \Leftrightarrow R \vdash_{\text{SXRC}} e == e'.$$

Proof. (i) $\Rightarrow$: By Propositions 71 and 69 we have

$$[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = \bigsqcup\{[\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \mid R \vdash_{\text{SXRC}} e \to t, t \in LT_\Sigma(X) \text{ simple}\}$$

and since $(\mathscr{CT}_{\Sigma,X}(R), id) \models e \to t$ it holds $[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsupseteq [\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = \mathbf{a}(t)$. Given that $[\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$ is finite, there must exist some t_0, such that

$$t_0 \in \{[\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \mid R \vdash_{\mathrm{SXRC}} e \to t, t \in LT_\Sigma(X) \text{ simple}\}$$

and $[\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsubseteq [\![t_0]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$. By Proposition 70, it holds $R \vdash_{\mathrm{SXRC}} t_0 \to t$, and, applying rule (T), it also holds $R \vdash_{\mathrm{SXRC}} e \to t$.

(ii) $\Rightarrow$: If $(\mathscr{CT}_{\Sigma,X}(R), id) \models e == e'$, then by Definition 56, there exists a totally defined element a, such that $a = [\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$, and $a = [\![e']\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$. Since a has finite domain, then $\mathbf{t}(a)$ is a simple labeled term. Now, by Proposition 69, $[\![\mathbf{t}(a)]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = \mathbf{a}(\mathbf{t}(a)) = a$, hence

$$[\![\mathbf{t}(a)]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsubseteq [\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{and} \quad [\![\mathbf{t}(a)]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsubseteq [\![e']\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$$

and, by Definition 56, it holds

$$(\mathscr{CT}_{\Sigma,X}(R), id) \models e \to \mathbf{t}(a) \quad \text{and} \quad (\mathscr{CT}_{\Sigma,X}(R), id) \models e' \to \mathbf{t}(a)$$

and by (i) $\Rightarrow$, $R \vdash_{\mathrm{SXRC}} e \to \mathbf{t}(a)$ and $R \vdash_{\mathrm{SXRC}} e' \to \mathbf{t}(a)$. Finally, applying the rule (J) we obtain $R \vdash_{\mathrm{SXRC}} e == e'$.

(i) $\Leftarrow$: We have that $R \vdash_{\mathrm{SXRC}} e \to t$, with $t \in LT_\Sigma(X)$ simple. By Propositions 71, and 69

$$[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = \bigsqcup\{[\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \mid R \vdash_{\mathrm{SXRC}} e \to t, t \in LT_\Sigma(X) \text{ simple}\},$$

that is $[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsupseteq [\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$, and by Definition 56, $(\mathscr{CT}_{\Sigma,X}(R), id) \models e \to t$.

(ii) $\Leftarrow$: We have that $R \vdash_{\mathrm{SXRC}} e == e'$, which can only be obtained by an application of rule (J) of SXRC. That means we have $R \vdash_{\mathrm{SXRC}} e \to t$, and $R \vdash_{\mathrm{SXRC}} e' \to t$, where $t \in LT_\Sigma(V)$ simple without $\bot$. By ((i) $\Leftarrow$) and Definition 56, it holds

$$[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsupseteq [\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{and} \quad [\![e']\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsupseteq [\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$$

Since id is totally defined, by Proposition 47, we have that $[\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$ is finite and total, having

$$[\![e]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = [\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{and} \quad [\![e']\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = [\![t]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$$

for being maximal. Finally, by Definition 56, $(\mathscr{CT}_{\Sigma,X}(R), id) \models e == e'$. □

To extend the characterization lemma to any given assignment we consider the effect of assignments as substitutions. We say that α, a $\mathscr{CT}_{\Sigma,X}(R)$ assignment for X, has *finite range* if $\alpha(x)$ is finite for any $x \in X$. Finite assignments can be seen as substitutions if we considered the associated simple labeled term, $\bar{\alpha}(x) =_{\mathrm{def}} \mathbf{t}(\alpha(x))$, for any $x \in X$. Observe that $\bar{\alpha}$ is a simple substitution.

Corollary 73. *Given an XGOTA specification $E = (\Sigma, R)$, where R is deterministic, any set of variables $X \subseteq V$, and a $\mathscr{CT}_{\Sigma,X}(R)$-assignment for X, α with finite range*

(i) *For any reduction statement* $e \to t$, *with* $e \in LE_\Sigma(X)$, $t \in LT_\Sigma(X)$ *simple*,

$$(\mathscr{CT}_{\Sigma,X}(R), \alpha) \models e \to t \Leftrightarrow R \vdash_{\mathrm{SXRC}} e\bar{\alpha} \to t\bar{\alpha}$$

(ii) *For any reduction statement* $e \to e'$, *with* $e, e' \in LE_\Sigma(X)$,

$$(\mathscr{CT}_{\Sigma,X}(R), \alpha) \models e \to e' \Leftarrow R \vdash_{\mathrm{SXRC}} e\bar{\alpha} \to e'\bar{\alpha}$$

(iii) *For any strict equation* $e == e'$, *with* $e, e' \in LE_\Sigma(X)$,

$$(\mathscr{CT}_{\Sigma,X}(R), \alpha) \models e == e' \Leftrightarrow R \vdash_{\mathrm{SXRC}} e\bar{\alpha} == e'\bar{\alpha}$$

Proof. (i) For any reduction statement $e \to t$, with $e \in LE_\Sigma(X)$, $t \in LT_\Sigma(X)$ simple,

$$\begin{aligned}
&(\mathscr{CT}_{\Sigma,X}(R), \alpha) \models e \to t \\
&\quad \Leftrightarrow [\![e]\!]_\alpha^{\mathscr{CT}_{\Sigma,X}(R)} \sqsupseteq [\![t]\!]_\alpha^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{(Definition 56)} \\
&\quad \Leftrightarrow [\![e\bar{\alpha}]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \sqsupseteq [\![t\bar{\alpha}]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{(Lemma 48)} \\
&\quad \Leftrightarrow (\mathscr{CT}_{\Sigma,X}(R), id) \models e\bar{\alpha} \to t\bar{\alpha} \quad \text{(Definition 56)} \\
&\quad \Leftrightarrow R \vdash_{\mathrm{SXRC}} e\bar{\alpha} \to t\bar{\alpha} \quad \text{(Characterization Lemma)}
\end{aligned}$$

Observe that $t\bar{\alpha} \in LT_\Sigma(X)$ simple, because $\bar{\alpha}$ is simple.

(ii) For any reduction statement $e \to e'$, with $e, e' \in LE_\Sigma(X)$, it holds

$$\begin{aligned}
[\![e']\!]_\alpha^{\mathscr{CT}_{\Sigma,X}(R)} &= [\![e'\bar{\alpha}]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{(Lemma 48)} \\
&= \bigsqcup\{\mathbf{a}(t) \mid R \vdash_{\mathrm{SXRC}} e'\bar{\alpha} \to t, t \in LT_\Sigma(X) \text{ simple}\} \quad \text{(Proposition 71)} \\
&\sqsubseteq \bigsqcup\{\mathbf{a}(t) \mid R \vdash_{\mathrm{SXRC}} e\bar{\alpha} \to t, t \in LT_\Sigma(X) \text{ simple}\} \quad (R \vdash_{\mathrm{SXRC}} e\bar{\alpha} \to e'\bar{\alpha}) \\
&= [\![e\bar{\alpha}]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{(Proposition 71)} \\
&= [\![e]\!]_\alpha^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{(Lemma 48)}
\end{aligned}$$

hence having that $(\mathscr{CT}_{\Sigma,X}(R), \alpha) \models e \to e'$.

(iii) For any strict equation $e == e'$, with $e, e' \in LE_\Sigma(X)$,

$$\begin{aligned}
&(\mathscr{CT}_{\Sigma,X}(R), \alpha) \models e == e' \\
&\Leftrightarrow [\![e]\!]_\alpha^{\mathscr{CT}_{\Sigma,X}(R)} = a = [\![e']\!]_\alpha^{\mathscr{CT}_{\Sigma,X}(R)}, && \text{for some } a \in \mathbf{tot}(\mathscr{C}_{\mathscr{CT}_{\Sigma,X}(R)}) \\
& && \text{(Definition 56)} \\
&\Leftrightarrow [\![e\bar{\alpha}]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} = a = [\![e'\bar{\alpha}]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}, && \text{for some } a \in \mathbf{tot}(\mathscr{C}_{\mathscr{CT}_{\Sigma,X}(R)}) \\
& && \text{(Lemma 48)} \\
&\Leftrightarrow (\mathscr{CT}_{\Sigma,X}(R), id) \models e\bar{\alpha} == e'\bar{\alpha} && \text{(Definition 56)} \\
&\Leftrightarrow R \vdash_{\mathrm{SXRC}} e\bar{\alpha} == e'\bar{\alpha} && \text{(Characterization Lemma).} \quad \square
\end{aligned}$$

Now, we can show that $\mathscr{CT}_{\Sigma,X}(R)$ is a model of an XGOTA specification (Σ,R), as long as R is deterministic.

Theorem 74. *Given an XGOTA specification $E=(\Sigma,R)$, where R is deterministic, and a set of variables $X\subseteq V$, the Σ-algebra of canonical trees by R, $\mathscr{CT}_{\Sigma,X}(R)$ is a model of E, $\mathscr{CT}_{\Sigma,X}(R)\models E$.*

Proof (*Sketch*). By Theorem 68, $\mathscr{CT}_{\Sigma,X}(R)$ is an XGOTA algebra for Σ. Now, we only have to prove that $\mathscr{CT}_{\Sigma,X}(R)$ satisfies every hierarchy form of R. The trouble is that we cannot apply Corollary 73 directly because it only deals with finite range assignments. The solution is to consider *finite approximations*, $\alpha\|_i$ to assignments α, defined by taking the pruning of the values assigned to every variable. The set of approximations so defined is an increasing chain, and it holds $(\bigsqcup_{i\in\mathbb{N}}\alpha\|_i)=\alpha$. Moreover, when it comes to condition satisfaction under an assignment α, there exists a significant index from which on, every approximation behaves as the original assignment. When proving the satisfaction of a given hierarchy form, whether from standard or auto rules, as we deal with least upper bounds for denotations we can forget everything under the significant index. Then we can apply Corollary 73 to finite range assignment approximations to obtain the satisfiability of the hierarchy form. Details can be found in [32]. □

The following proposition gathers the results with respect to $\mathscr{CT}_{\Sigma,X}(R)$.

Proposition 75. *Given an XGOTA specification $E=(\Sigma,R)$, where R is deterministic, and a set of variables X, the following statements are equivalent*:

(i) $R\vdash_{\mathrm{SXRC}}\phi$,

(ii) $(\mathscr{A},\alpha)\models\phi$, *for every* $\mathscr{A}\models E$, *and* α *totally defined* $\mathscr{A}$*-assignment for* X,

(iii) $(\mathscr{CT}_{\Sigma,X}(R),id)\models\phi$,

where ϕ is any reduction statement $e\to t$, or strict equation $e==e'$, with $e,\ e'\in LE_\Sigma(X)$, $t\in LT_\Sigma(X)$ simple.

Proof. Statement (i) $\Rightarrow$ (ii) holds from the SXRC Soundness Theorem, (ii) $\Rightarrow$ (iii) holds from Theorem 74, because *id* is a totally defined $\mathscr{CT}_{\Sigma,X}(R)$-assignment, and (iii) $\Rightarrow$ (i) holds from the Characterization Lemma. □

Theorem 76 (SXRC completeness). *Given an XGOTA specification $E=(\Sigma,R)$,*

(i) *For any reduction statement $e\to t$, with $e\in LE_\Sigma(X)$, $t\in LT_\Sigma(X)$ simple, if $(\mathscr{A},\alpha)\models e\to t$, for every $\mathscr{A}\models E$, and α totally defined $\mathscr{A}$-assignment for X then $R\vdash_{\mathrm{SXRC}} e\to t$.*

(ii) *For any strict equation $e==e'$, with $e,e'\in LE_\Sigma(X)$, if $(\mathscr{A},\alpha)\models e==e'$, for every $\mathscr{A}\models E$, and α totally defined $\mathscr{A}$-assignment for X then $R\vdash_{\mathrm{SXRC}} e==e'$.*

Proof. Follows from Proposition 75, when X includes the set of variables occurring in $e\to t$, or $e==e'$. □

8. Initial semantics

Among the many possible models of a given XGOTA specification, we are interested in characterizing those models that do not introduce any unspecified objects or behavior. As in [34, 20], the point is to have a *standard* model which has the minimal number of abstract values (none that cannot be constructed from the permitted attribute combinations), and satisfies the minimal number of reduction statements, and strict equalities (none that do not follow from hierarchy forms). We intend to characterize them as *initial models* [18], which show a special relation with the other models of a specification. To express that relation we need the concept of *homomorphism* between XGOTA algebras.

Definition 77. Given an extended signature $\Sigma = (S, \leqslant, A, F)$, and two extended Σ-algebras $\mathscr{A}$ and $\mathscr{B}$, we define a *homomorphism* between $\mathscr{A}$ and $\mathscr{B}$ noted by $h : \mathscr{A} \to \mathscr{B}$, as a continuous mapping h between the carrier sets of $\mathscr{A}$ and $\mathscr{B}$, $h : \mathscr{C}_{\mathscr{A}} \to \mathscr{C}_{\mathscr{B}}$ such that

(1) $h(\perp_{\mathscr{A}}) = \perp_{\mathscr{B}}$ (h is strict).
(2) $h((s\{G\})^{\mathscr{A}}) \subseteq (s\{G\})^{\mathscr{B}}$ for every genome $s\{G\}$ of Σ.
(3) $h(c_s^{\mathscr{A}}(a_1, \ldots, a_n)) = c_s^{\mathscr{B}}(h(a_1), \ldots, h(a_n))$ for every sort s of Σ with n attributes ($n > 0$), and every value $a_1 \in \mathscr{C}_{\mathscr{A}}, \ldots, a_n \in \mathscr{C}_{\mathscr{A}}$ such that there exists an ordered complete set of attributes of s, $g_i : s_i \to w_i \in A$ with $a_i \in w_i^{\mathscr{A}}$, for $i = 1, \ldots, n$.
(4) $h(c_s^{\mathscr{A}}) = c_s^{\mathscr{B}}$ for every sort s of S with neither attributes, nor subsorts.
(5) $h(f_d^{\mathscr{A}}(a_1, \ldots, a_n)) \sqsubseteq f_d^{\mathscr{B}}(h(a_1), \ldots, h(a_n))$ for every declaration d, of an operation symbol f with domain $s_1\{G_1\} \cdots s_n\{G_n\}$, and every value $a_1 \in (s_1\{G_1\})^{\mathscr{A}}, \ldots, a_n \in (s_n\{G_n\})^{\mathscr{A}}$.
(6) $h(f_d^{\mathscr{A}}) \sqsubseteq f_d^{\mathscr{B}}$ for every declaration d, of an operation symbol f with no arguments.

The existence of a single way of mapping the elements of an algebra with those of any other, characterizes a standard model, that is defined uniquely up to renaming algebra elements [34]. It is the property of an algebra of being *initial* in the class $\mathrm{XAlg}_{\Sigma}(R)$ of all XGOTA Σ-algebras which are models of $E = (\Sigma, R)$.

Definition 78. Given an XGOTA signature Σ, and an XGOTA Σ-algebra $\mathscr{A}$, we say that $\mathscr{A}$ is an *initial algebra*, or just *initial*, if for any XGOTA Σ-algebra $\mathscr{B}$, there exists exactly one homomorphism h between $\mathscr{A}$ and $\mathscr{B}$.

To show that the algebra of canonical trees is initial we will need to construct the homomorphism h whose existence and uniqueness is required by Definition 78. What we are going to do is to prove a stronger result, specifically, we will show that, given a deterministic Σ-program R, the algebra of canonical trees by R, $\mathscr{CT}_{\Sigma,X}(R)$ is the *free algebra* generated by the set of variables X.

Definition 79. Given a signature Σ, an extended Σ-algebra $\mathscr{A}$, and a set of variables X, we say that $\mathscr{A}$ is the *free algebra* generated by the set of variables X, if for any

extended Σ-algebra $\mathscr{B}$, and any totally defined $\mathscr{B}$-assignment for X, α, there exists exactly one homomorphism h between $\mathscr{A}$ and $\mathscr{B}$ that extends α.

In the next proposition we show that the canonical tree model $\mathscr{CT}_{\Sigma,X}(R)$ is free. In the particular case $X=\varnothing$, freeness just means initiality.

Proposition 80. *Given an XGOTA specification $E=(\Sigma,R)$, where R is deterministic, and a set of variables X, the algebra of canonical trees by R, $\mathscr{CT}_{\Sigma,X}(R)$ is freely generated by X in the class* $\mathrm{XAlg}_{\Sigma}(R)$ *of all models of $E=(\Sigma,R)$.*

Proof. We have to show that for any model of E, $\mathscr{A}$, and any totally defined $\mathscr{A}$-assignment for X, α, there exists exactly one homomorphism $h:\mathscr{CT}_{\Sigma,X}(R)\to\mathscr{A}$ which extends α. That means proving five statements:

(i) *There exists a mapping $h:CT_{\Sigma}(X)\to\mathscr{C}_{\mathscr{A}}$.* For every $a\in CT_{\Sigma}(X)$ we define $h(a)$ as the meaning in $\mathscr{A}$ of the corresponding abstract value. If $(\mathscr{C}_{\mathscr{A}},\sqsubseteq_{\mathscr{A}},\bot_{\mathscr{A}})$ is the CPO carrier set of $\mathscr{A}$, we can define

$$h(a)=_{\mathrm{def}}\bigsqcup_{k\in\mathbb{N}}[\![\mathbf{t}(a\|_k)]\!]^{\mathscr{A}}_{\alpha}.$$

Observe that we have to use the least upper bound of the associated terms of the prunings of tree a to obtain its corresponding meaning, for the case in which a is an infinite tree. This definition is correct since $K=\{[\![\mathbf{t}(a\|_k)]\!]^{\mathscr{A}}_{\alpha}\}_{k\in\mathbb{N}}$ is an increasing chain, which trivially implies that K is directed. This can easily be proven by induction on k. Therefore, h is correctly defined for any $a\in CT_{\Sigma}(X)$.

(ii) *h is continuous.* Note that if

$$\left(\bigsqcup_{i\in I}a_i\right)\|_k=\bigsqcup_{i\in I}(a_i\|_k) \tag{2}$$

for any directed set $\{a_i\}_{i\in I}$ in $(CT_{\Sigma}(X),\sqsubseteq,\bot)$, and

$$\left[\!\!\left[\mathbf{t}\left(\bigsqcup_{i\in I}b_i\right)\right]\!\!\right]^{\mathscr{A}}_{\alpha}=\bigsqcup_{i\in I}[\![\mathbf{t}(b_i)]\!]^{\mathscr{A}}_{\alpha} \tag{3}$$

with $\bigsqcup_{i\in I}b_i$, and b_i finite in $CT_{\Sigma}(X)$, for $i\in I$, hold, then

$$\begin{aligned}
h\left(\bigsqcup_{i\in I}a_i\right)&=\bigsqcup_{k\in\mathbb{N}}[\![\mathbf{t}\left(\left(\bigsqcup_{i\in I}a_i\right)\|_k\right)]\!]^{\mathscr{A}}_{\alpha}\quad\text{(Definition of } h\text{)}\\
&=\bigsqcup_{k\in\mathbb{N}}[\![\mathbf{t}\left(\bigsqcup_{i\in I}(a_i\|_k)\right)]\!]^{\mathscr{A}}_{\alpha}\quad\text{(Eq. (2))}\\
&=\bigsqcup_{k\in\mathbb{N}}\bigsqcup_{i\in I}[\![\mathbf{t}(a_i\|_k)]\!]^{\mathscr{A}}_{\alpha}\quad\text{(Eq. (3))}\\
&=\bigsqcup_{i\in I}\bigsqcup_{k\in\mathbb{N}}[\![\mathbf{t}(a_i\|_k)]\!]^{\mathscr{A}}_{\alpha}\\
&=\bigsqcup_{i\in I}h(a_i)\quad\text{(Definition of } h\text{)}.
\end{aligned}$$

Observe that Eq. (2) holds trivially since we have defined the least upper bound in $(CT_\Sigma(X), \sqsubseteq, \bot)$ as joins, and pruning the join is like joining the prunings. Eq. (3) can be shown by structural induction over $\bigsqcup_{i\in I} b_i$. The cases when $\bigsqcup_{i\in I} b_i$ is $\bot$, or a variable are simple. When it is a feature term, the proof relies on the fact that denotations are obtained with continuous mappings, while for attribute values it holds $(\bigsqcup_{i\in I} b_i)|_g = \bigsqcup_{i\in I}(b_i|_g)$ because restricting the join is like joining the restrictions.

(iii) *h is a homomorphism*: We have to show that the continuous mapping h defined in step (i) fulfills the requirements of the definition of homomorphism:

(1) By definition of h we have that

$$h(\bot) = \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(\bot\|_k)]\!]_\alpha^{\mathscr{A}} = \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(\bot)]\!]_\alpha^{\mathscr{A}} = \bigsqcup_{k\in\mathbb{N}} [\![\bot]\!]_\alpha^{\mathscr{A}} = [\![\bot]\!]_\alpha^{\mathscr{A}} = \bot_{\mathscr{A}}.$$

(2) We have to show that for every $a \in (s\{G\})^{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}$ it holds that $h(a) \in (s\{G\})^{\mathscr{A}}$. By definition of $\mathscr{C}\mathscr{T}_{\Sigma,X}(R)$ we have that $(s\{G\})^{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}$ is the set of canonical trees with genome $s\{G\}$. Now, for any $a \in (s\{G\})^{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}$, $\mathbf{t}(a\|_k)$ is a term with genome $s\{G\}$. Therefore, by Proposition 37, $[\![\mathbf{t}(a\|_k)]\!]_\alpha^{\mathscr{A}} \in (s\{G\})^{\mathscr{A}}$, and $\bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a\|_k)]\!]_\alpha^{\mathscr{A}} \in (s\{G\})^{\mathscr{A}}$, since $(s\{G\})^{\mathscr{A}}$ is a CPO by Definition 32, and $h(a) \in (s\{G\})^{\mathscr{A}}$.

(3) Let $s \in S$ be a sort with n attributes $(n>0)$, where $g_1,\ldots,g_n$ is the ordered set of attributes of s, and $a_1 \in \mathscr{C}_{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}, \ldots, a_n \in \mathscr{C}_{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}$, such that there exists an ordered complete set of attributes of s, $g_i : s_i \to w_i \in A$ with $a_i \in w_i^{\mathscr{A}}$, for $i = 1,\ldots,n$. We have to show that $h(c_s^{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}(a_1,\ldots,a_n)) = c_s^{\mathscr{A}}(h(a_1),\ldots,h(a_n))$, which holds by

$$\begin{aligned}
&h(c_s^{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}(a_1,\ldots,a_n))\\
&\quad= h\left(\bigcup_{i=1,\ldots,n} a_i\,|^{g_i}(s)\right) \quad \text{(Definition 66)}\\
&\quad= \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}\left(\left(\bigcup_{i=1,\ldots,n} a_i\,|^{g_i}(s)\right)\|_k\right)]\!]_\alpha^{\mathscr{A}} \quad \text{(Definition of } h\text{)}\\
&\quad= \bigsqcup_{k>0} [\![\mathbf{t}\left(\left(\bigcup_{i=1,\ldots,n} a_i\,|^{g_i}(s)\right)\|_k\right)]\!]_\alpha^{\mathscr{A}} \quad (K \text{ is an increasing chain})\\
&\quad= \bigsqcup_{k>0} [\![s[g_1 \Rightarrow \mathbf{t}(a_1\|_{(k-1)}),\ldots,g_n \Rightarrow \mathbf{t}(a_n\|_{(k-1)})]]\!]_\alpha^{\mathscr{A}} \quad \text{(Definition 65)}\\
&\quad= \bigsqcup_{k>0} c_s^{\mathscr{A}}([\![\mathbf{t}(a_1\|_{(k-1)})]\!]_\alpha^{\mathscr{A}},\ldots,[\![\mathbf{t}(a_n\|_{(k-1)})]\!]_\alpha^{\mathscr{A}}) \quad \text{(Definition 36)}\\
&\quad= c_s^{\mathscr{A}}\left(\bigsqcup_{k>0} [\![\mathbf{t}(a_1\|_{(k-1)})]\!]_\alpha^{\mathscr{A}},\ldots,\bigsqcup_{k>0} [\![\mathbf{t}(a_n\|_{(k-1)})]\!]_\alpha^{\mathscr{A}}\right) \quad (c_s^{\mathscr{A}} \text{ continuous})
\end{aligned}$$

$$= c_s^{\mathscr{A}}\left(\bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a_1\|_k)]\!]_\alpha^{\mathscr{A}}, \ldots, \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a_n\|_k)]\!]_\alpha^{\mathscr{A}}\right)$$

$$= c_s^{\mathscr{A}}(h(a_1), \ldots, h(a_n)) \quad \text{(Definition of } h\text{)}.$$

(4) We have to show $h(c_s^{\mathscr{CT}_{\Sigma,X}(R)}) = c_s^{\mathscr{A}}$, for every sort s of S with neither attributes, nor subsorts. By Definition 66, $c_s^{\mathscr{CT}_{\Sigma,X}(R)}$ is the canonical tree a, such that $D_a = \{\varepsilon\}$ and $a(\varepsilon) = s$, but a is a 0-level tree, which means that $a\|_k \equiv a$ for any $k > 0$, and since K is an increasing chain

$$h(a) = \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{a}(a\|_k)]\!]_\alpha^{\mathscr{A}} = \bigsqcup_{k>0} [\![\mathbf{a}(a)]\!]_\alpha^{\mathscr{A}} = \bigsqcup_{k>0} [\![s[]]\!]_\alpha^{\mathscr{A}} = [\![s[]]\!]_\alpha^{\mathscr{A}} = c_s^{\mathscr{A}}$$

hence having the result desired.

(5) Let us consider $a_1 \in (s_1\{G_1\})^{\mathscr{CT}_{\Sigma,X}(R)}, \ldots, a_n \in (s_n\{G_n\})^{\mathscr{CT}_{\Sigma,X}(R)}$. Note that if

$$\bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1, \ldots, a_n)\|_k)]\!]_\alpha^{\mathscr{A}} \sqsubseteq \bigsqcup_{k\in\mathbb{N}} [\![d : f(\mathbf{t}(a_1\|_k), \ldots, \mathbf{t}(a_n\|_k))]\!]_\alpha^{\mathscr{A}} \tag{4}$$

holds, then

$$h(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1, \ldots, a_n)) = \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1, \ldots, a_n)\|_k)]\!]_\alpha^{\mathscr{A}} \quad \text{(Definition of } h\text{)}$$

$$\sqsubseteq \bigsqcup_{k\in\mathbb{N}} [\![d : f(\mathbf{t}(a_1\|_k), \ldots, \mathbf{t}(a_n\|_k))]\!]_\alpha^{\mathscr{A}} \quad \text{(Eq. (4))}$$

$$= \bigsqcup_{k\in\mathbb{N}} f_d^{\mathscr{A}}([\![\mathbf{t}(a_1\|_k)]\!]_\alpha^{\mathscr{A}}, \ldots, [\![\mathbf{t}(a_n\|_k)]\!]_\alpha^{\mathscr{A}}) \quad \text{(Definition 44)}$$

$$= f_d^{\mathscr{A}}\left(\bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a_1\|_k)]\!]_\alpha^{\mathscr{A}}, \ldots, \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a_n\|_k)]\!]_\alpha^{\mathscr{A}}\right)$$

($f_d^{\mathscr{A}}$ is continuous)

$$= f_d^{\mathscr{A}}(h(a_1), \ldots, h(a_n)) \quad \text{(Definition of } h\text{)}.$$

To show Eq. (4), we can prove that for any $l \in \mathbb{N}$, there exists some $l' \in \mathbb{N}$, such that

$$[\![\mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1, \ldots, a_n)\|_l)]\!]_\alpha^{\mathscr{A}} \sqsubseteq [\![d : f(\mathbf{t}(a_1\|_{l'}), \ldots, \mathbf{t}(a_n\|_{l'}))]\!]_\alpha^{\mathscr{A}}$$

and to show this result we consider some $l \in \mathbb{N}$. By Proposition 64, it holds $a = \bigsqcup_{k\in\mathbb{N}} a\|_k$, for any canonical tree a, hence

$$f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1, \ldots, a_n) = f_d^{\mathscr{CT}_{\Sigma,X}(R)}\left(\bigsqcup_{k\in\mathbb{N}} a_1\|_k, \ldots, \bigsqcup_{k\in\mathbb{N}} a_n\|_k\right)$$

$$= \bigsqcup_{k\in\mathbb{N}} f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1\|_k, \ldots, a_n\|_k) \quad (f_d^{\mathscr{CT}_{\Sigma,X}(R)} \text{ is continuous}).$$

Now, since $(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n))\|_l$ is finite, and

$$f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l \sqsubseteq \bigsqcup_{k\in\mathbb{N}} f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_k$$
$$= f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)$$
$$= \bigsqcup_{k\in\mathbb{N}} f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1\|_k,\ldots,a_n\|_k)$$

being an algebraic CPO, there must exist some $l'\in\mathbb{N}$, such that

$$f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l \sqsubseteq f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1\|_{l'},\ldots,a_n\|_{l'}).$$

Therefore, it holds

$$[\![\mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l)]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}$$
$$= \mathbf{a}(\mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l)) \quad \text{(Proposition 69)}$$
$$= f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l$$
$$\sqsubseteq f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1\|_{l'},\ldots,a_n\|_{l'})$$
$$= f_d^{\mathscr{CT}_{\Sigma,X}(R)}(\mathbf{a}(\mathbf{t}(a_1\|_{l'})),\ldots,\mathbf{a}(\mathbf{t}(a_n\|_{l'})))$$
$$= f_d^{\mathscr{CT}_{\Sigma,X}(R)}([\![\mathbf{t}(a_1\|_{l'})]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)},\ldots,[\![\mathbf{t}(a_n\|_{l'})]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)}) \quad \text{(Proposition 69)}$$
$$= [\![d : f(\mathbf{t}(a_1\|_{l'}),\ldots,\mathbf{t}(a_n\|_{l'}))]\!]_{id}^{\mathscr{CT}_{\Sigma,X}(R)} \quad \text{(Definition 44)}$$

and by Definition 56, it holds

$$(\mathscr{CT}_{\Sigma,X}(R), id) \models d : f(\mathbf{t}(a_1\|_{l'}),\ldots,\mathbf{t}(a_n\|_{l'})) \to \mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l)$$

Now, by Proposition 75, given that $\mathscr{A}$ is a model, and α is totally defined,

$$(\mathscr{A},\alpha) \models d : f(\mathbf{t}(a_1\|_{l'}),\ldots,\mathbf{t}(a_n\|_{l'})) \to \mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l)$$

and, again by Definition 56, it holds

$$[\![\mathbf{t}(f_d^{\mathscr{CT}_{\Sigma,X}(R)}(a_1,\ldots,a_n)\|_l)]\!]_\alpha^{\mathscr{A}} \sqsubseteq [\![d : f(\mathbf{t}(a_1\|_{l'}),\ldots,\mathbf{t}(a_n\|_{l'}))]\!]_\alpha^{\mathscr{A}}$$

which is the result we were seeking.

(6) This step is formally analogous to (5).

(iv) *h extends* α: We can define the immersion mapping $i : xX \to \mathscr{CT}_{\Sigma,X}(R)$ such that for any variable $x\in X$, $i(x)$ is $\mathbf{a}(x)$. (Observe that i coincides with id.) With the definition of h, and since K is an increasing chain, and $\mathbf{a}(x)$ is a 0-level tree,

we have that

$$h(i(x)) = h(\mathbf{a}(x)) = \bigsqcup_{k>0} [\![\mathbf{t}(\mathbf{a}(x)\|_k)]\!]_\alpha^{\mathscr{A}} = \bigsqcup_{k>0} [\![\mathbf{t}(\mathbf{a}(x))]\!]_\alpha^{\mathscr{A}} = [\![x]\!]_\alpha^{\mathscr{A}} = \alpha(x)$$

having that h extends α.

(v) *There is only one such h*: We will show that any other homomorphism $h' : CT_\Sigma(X) \to \mathscr{C}_{\mathscr{A}}$ coincides with h. We will prove $h'(a) = h(a)$ for every canonical tree $a \in CT_\Sigma(X)$ in two steps. Firstly, for any finite tree a by structural induction over a:

- If a is $\bot$, then by Definition 77, since h and h' are homomorphisms

$$h(a) = h(\bot) = \bot_{\mathscr{A}} = h'(\bot) = h'(a),$$

  hence having that $h(a) = h'(a)$.
- If a is such that $a(\varepsilon) = x \in X$, then $D_a = \{\varepsilon\}$, and since h and h' must extend α it must hold

$$h(a) = h(\mathbf{a}(x)) = h(i(x)) = \alpha(x) = h'(i(x)) = h'(\mathbf{a}(x)) = h'(a),$$

  hence having that $h(a) = h'(a)$.
- If a is such that $a(\varepsilon) = w \in S$, a sort with ordered set of attributes $\langle g_1, \ldots, g_n \rangle$ $(n \geqslant 0)$, where $a_i = a|_{g_i}$ for $i = 1, \ldots, n$, then

$$\begin{aligned}
h(a) &= \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a\|_k)]\!]_\alpha^{\mathscr{A}} \quad \text{(Definition of } h\text{)} \\
&= \bigsqcup_{k>0} [\![\mathbf{t}(a\|_k)]\!]_\alpha^{\mathscr{A}} \quad (K \text{ is an increasing chain}) \\
&= \bigsqcup_{k>0} [\![w[g_1 \Rightarrow \mathbf{t}(a_1\|_{(k-1)}), \ldots, g_n \Rightarrow \mathbf{t}(a_n\|_{(k-1)})]]\!]_\alpha^{\mathscr{A}} \quad \text{(Definition 63)} \\
&= \bigsqcup_{k>0} c_w^{\mathscr{A}}([\![\mathbf{t}(a_1\|_{(k-1)})]\!]_\alpha^{\mathscr{A}}, \ldots, [\![\mathbf{t}(a_n\|_{(k-1)})]\!]_\alpha^{\mathscr{A}}) \quad \text{(Definition 36)} \\
&= c_w^{\mathscr{A}}\left(\bigsqcup_{k>0} [\![\mathbf{t}(a_1\|_{(k-1)})]\!]_\alpha^{\mathscr{A}}, \ldots, \bigsqcup_{k>0} [\![\mathbf{t}(a_n\|_{(k-1)})]\!]_\alpha^{\mathscr{A}}\right) \quad (c_w^{\mathscr{A}} \text{ is continuous}) \\
&= c_w^{\mathscr{A}}\left(\bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a_1\|_k)]\!]_\alpha^{\mathscr{A}}, \ldots, \bigsqcup_{k\in\mathbb{N}} [\![\mathbf{t}(a_n\|_k)]\!]_\alpha^{\mathscr{A}}\right) \\
&= c_w^{\mathscr{A}}(h(a_1), \ldots, h(a_n)) \quad \text{(Definition of } h\text{)} \\
&= c_w^{\mathscr{A}}(h'(a_1), \ldots, h'(a_n)) \quad \text{(Structural induction hypothesis)} \\
&= h'(c_w^{\mathscr{C}\mathscr{T}_{\Sigma,X}(R)}(a_1, \ldots, a_n)) \quad (h' \text{ is a homomorphism})
\end{aligned}$$

$$= h'\left(\bigcup_{i=1,\ldots,n} a_i \,|^{g_i}(s)\right) \quad \text{(Definition 66)}$$

$$= h'(a),$$

hence having $h(a) = h'(a)$.

Now, for any $a \in CT_\Sigma(X)$, it holds

$$h(a) = h\left(\bigsqcup_{a_0 \in \mathbf{B}(a)} a_0\right) = \bigsqcup_{a_0 \in \mathbf{B}(a)} h(a_0) = \bigsqcup_{a_0 \in \mathbf{B}(a)} h'(a_0) = h'\left(\bigsqcup_{a_0 \in \mathbf{B}(a)} a_0\right) = h'(a)$$

since h and h' are continuous and a_0 is finite. □

Now, we can show that there exists an initial model for the class of all XGOTA Σ-algebras which are models of $E = (\Sigma, R)$.

Corollary 81. *Given an XGOTA signature $\Sigma = (S, \leqslant, A)$ the algebra of ground canonical trees $\mathscr{CT}_\Sigma(R)$ is initial model in the class $\mathrm{XAlg}_\Sigma(R)$ of all XGOTA Σ-algebras which are models of $E = (\Sigma, R)$.*

Proof. By Theorem 68, we have that $\mathscr{CT}_{\Sigma,X}(R)$ is Σ-algebra for any set of variables X, and by Theorem 74 is a model of the specification $E = (\Sigma, R)$. In particular, if we consider the empty set of variables, we get the algebra of ground canonical trees $\mathscr{CT}_\Sigma(R)$ which is a model as well. By Proposition 80, for the empty assignment of variables, there exists exactly one homomorphism between $\mathscr{CT}_\Sigma(R)$ and any other algebra $\mathscr{A} \in \mathrm{XAlg}_\Sigma(R)$, this implies that $\mathscr{CT}_\Sigma(R)$ is initial in $\mathrm{XAlg}_\Sigma(R)$. □

9. Conclusions and further work

We have presented XGOTA, a framework to model object orientation within an algebraic setting. Classes, inheritance, attributes and methods play a primitive role while they are given a declarative semantics. As we sought, the representation distance is fairly reduced with respect to other formal models of object oriented programming. In particular, our framework allows to formalize methods by rewrite rules, which leads to more a natural formulation in comparison to approaches based on λ-like calculi.

Our proposal is based on existing ideas, such as modelling object states by feature terms and class hierarchies with the help of order-sorted signatures. However, we have introduced new concepts like genetic inheritance, genome typing, an homogeneous treatment of methods and class-external procedures as algebraic operations, and auto rewriting rules for mutator methods. We have shown the power of genetic inheritance to model objects with mutable features, which turn out to be helpful to control execution. This power relies on a liberal overriding policy which does not impose the covariance conditions found in some other approaches. The price we pay for this freedom is

the loss of dynamic dispatch: in order to preserve well-typed expressions along a rewriting computation, we must stick to a static type discipline. This limitation cannot be overcome in our current framework, although it could be avoided for a restricted subset of XGOTA specifications with a more restrictive overriding policy. Looking for alternative semantics that might allow to support dynamic dispatch while keeping a liberal overriding policy, is a matter of future research.

Object-oriented environments and languages enjoy certain operative characteristics related with concurrency like object creation and deletion, or message passing. These features are not present in XGOTA as we deem them more related with execution rather than design of object specifications. We are currently working on developing a functional programming language with the object-oriented features of XGOTA, where the operational concepts which the theoretical framework lacks would be programmed within the framework itself. We plan to "code" concurrency with the help of classes for a pool of objects where creation and deletion of objects would be mutator methods, while message passing would rely on evaluation operations for that pool. The semantics for this language should be expressible as an XGOTA specification.

Acknowledgements

We would like to thank our colleagues Narciso Martí-Oliet, Antonio Gavilanes, and Juan Carlos González-Moreno for their comments. We specially appreciate the thorough work of the anonymous reviewers and their valuable comments.

References

[1] M. Abadi, L. Cardelli, A theory of primitive objects, Proc. Theoretical Aspects of Computer Software, Lecture Notes in Computer Science, Vol. 789, Springer, Berlin, 1994, pp. 296–320.
[2] M. Abadi, L. Cardelli, in: A Theory of Objects, Monographs in Computer Science Series, Springer, Berlin, 1996.
[3] M. Abadi, L. Cardelli, R. Viswanathan, An interpretation of objects and object types, Principles Programming Languages, Papers presented at the Symposium, St. Petersburg Beach, FL, USA, 21–24 January 1996. ACM Press, NY, 1996, pp. 396–409.
[4] H. Abelson, G.J. Sussman, J. Sussman, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, 1985.
[5] H. Aït-Kaci, A. Podelski, Functions as passive constraints, Tech. Report, Digital Paris Research Laboratory, November 1992.
[6] H. Aït-Kaci, A. Podelski, Towards a meaning of LIFE, J. Logic Programming 16 (1993) 195–234.
[7] H. Aït-Kaci, A. Podelski, G. Smolka, Feature-based constraint system for logic programming with entailment, Theoret. Comput. Sci. 122 (1–2) (1994) 263–283.
[8] J.M. Andreoli, R. Pareschi, Linear objects: logical processes with built-in inheritance, New Generation Comput. 9 (3–4) (1991) 445–473.
[9] K.B. Bruce, A paradigmatic object-oriented programming language: design, static typing and semantics, J. Funct. Programming 4 (2) (1994) 127–206.
[10] K.B. Bruce, L. Cardelli, B.C. Pierce, Comparing object encodings, TACS 1997 (415–438).
[11] L. Cardelli, A semantics of multiple inheritance, Inform. and Comput. 76 (2/3) (1988) 138–164.
[12] G. Castagna, G. Ghelli, G. Longo, A calculus for overloaded functions with subtyping, Inform. and Comput. 117 (1995) 115–135.

[13] A. Compagnoni, M. Fernández, in: An object calculus with algebraic rewriting, PLILP'97, Lecture Notes in Computer Science, 1292, Springer, Berlin, 1997.
[14] J.S. Conery, Logical objects, in: R.A. Kowalski, K.A. Bowen (Eds.), 5th Internat. Conf. Symp. on Logic Programming, 1988, pp. 420–434.
[15] G. Delzanno, M. Martelli, Objects in Forum, in: J. Lloyd (Ed.), Internat. Conf. Symp. on Logic Programming, MIT Press, Cambridge, MA, 1995, pp. 115–129.
[16] J. Goguen, R. Diaconescu, Towards an algebraic semantics for the object paradigm, 9th Workshop on Specification of Abstract Data Types, Lecture Notes in Computer Science, Col. 785, Springer, Berlin, 1992, pp. 1–29.
[17] J. Goguen, G. Malcolm, A hidden agenda, Oxford University Computing Laboratory, Programming Research Group, 1996.
[18] J. Goguen, J. Meseguer, Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations, Theoret. Comput. Sci. 105 (1992) 217–273.
[19] J. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, Initial algebra semantics and continuous algebras, J. ACM 24 (1) (1977) 68–95.
[20] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo, An approach to declarative programming based on a rewriting logic, J. Logic Programming 40 (1) (1999) 47–87.
[21] J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo, Denotational versus declarative semantics for functional programming, Proc. Internat. Conf. on Computer Science Logic (CSL'91), Lecture Notes in Computer Science, Vol. 626, Springer, Berlin, 1992, 134–148.
[22] C.A. Gunter, D. Scott, Semantic domains, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990, pp. 633–674 (Chapter 6).
[23] M. Henz, G. Smolka, J. Würtz, in: P. van Hentenryck, V. Saraswat (Eds.), Object-oriented concurrent constraint programming in Oz, Principles and Practice of Constraint Programming, MIT Press, Cambridge, MA, 1993.
[24] J.S. Hodas, D. Miller, Representing objects in a logic programming language with scoped constructs, in: D.H.D. Warren, P. Szeredi (Eds.), 7th Internat. Conf. on Logic Programming, MIT Press, Cambridge, MA, 1990, pp. 511–526.
[25] M. Hoffman, B.C. Pierce, A unifying type-theoretic framework for objects, J. Funct. Programming 5 (4) (1995) 593–635.
[26] B. Jacobs, in: B. Freitag, C.B. Jones, C. Lengauer, H.J. Schek (Eds.), Objects and classes, co-algebraically, Object-Orientation with Parallelism and Persistence, Kluwer Academic Publ., Dordrecht, 1996.
[27] B. Jacobs, Inheritance and cofree constructions, in: P. Cointe (Ed.), European Conf. on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 1098, Springer, Berlin, 1996, pp. 210–231.
[28] B. Jacobs, J. Rutten, A tutorial on (Co)algebras and (Co)induction, EATCS Bull. 62 (1997) 222–259.
[29] W.R. LaLonde, Designing families of data types using exemplars, ACM Trans. Programming Languages Systems 11 (2) (1989) 212–248.
[30] J. Mateos-Lago, M. Rodríguez-Artalejo, GOTA algebras: a specification formalism for inheritance and object hierarchies, PLILP'96, Lecture Notes in Computer Science, Vol. 1140, Springer, Berlin, 1996, pp. 62–76.
[31] J. Mateos Lago, M. Rodríguez Artalejo, Tagged feature terms and continuous GOTA algebras, Tech. Report 70/97, Departamento de Informática y Automática, Universidad Complutense de Madrid, 1997. (http://mozart.sip.ucm.es/).
[32] J. Mateos Lago, M. Rodríguez Artalejo, Operations with static typing in genetic inheritance object specifications, Tech. Report 82/98, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 1998. (http://mozart.sip.ucm.es/).
[33] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, A. Yonezawa (Eds.), Research Directions in Concurrent Object-Oriented Programming, The MIT Press, Cambridge, MA, 1993, pp. 315–390.
[34] J. Meseguer, J. Goguen, in: M. Nivat, J.C. Reynolds (Eds.), Initiality, induction and computability, Algebraic Methods in Semantics, Cambridge University Press, Cambridge, 1985.
[35] J. Meseguer, N. Martí-Oliet, From abstract data types to logical frameworks, in: E. Astesiano, G. Reggio, A. Tarlecki (Eds.), Recent Trends in Data Type Specification, Lecture Notes in Computer Science, Vol. 906, Springer, Berlin, 1995, pp. 48–80.

[36] L. Monteiro, A. Porto, Contextual logic programming, in: G. Levi, M. Martelli (Eds.), 6th Internat. Conf. on Logic Programming, The MIT Press, Cambridge, MA, 1989, pp. 284–299.
[37] B.C. Pierce, D.N. Turner, Simple type-theoretic foundations for object-oriented programming, J. Funct. Programming 4 (2) (1994) 107–247.
[38] J. Rumbaugh, M. Blaha, W. Premerlain, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Englewood cliffs, NJ, 1991.
[39] G. Smolka, The Oz programming model, in: J. Van Leeuwen (Ed.), Computer Science Today, Lecture Notes in Computer Science, Vol. 1000, Springer, Berlin, 1996, pp. 324–343.
[40] G. Smolka, H. Aït-Kaci, Inheritance hierarchies: semantics and unification, J. Symbolic Comput. 7 (1989) 343–370.
[41] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific J. Math. 5 (1955) 285–309.
[42] R.J. Wieringa, A formalization of objects using equational dynamic logic, in: C. Delobel, M. Kifer, Y. Masunaga (Eds.), 2nd Internat. Congress on Deductive and Object-Oriented Databases DOOD'91, Lecture Notes in Computer Science, Vol. 566, Springer, Berlin, 1991, pp. 431–452.